Blending Liquids

Karthik Raveendran* Georgia Tech
Chris Wojtan† IST Austria
Nils Thuerey‡ TU Munich
Greg Turk§ Georgia Tech

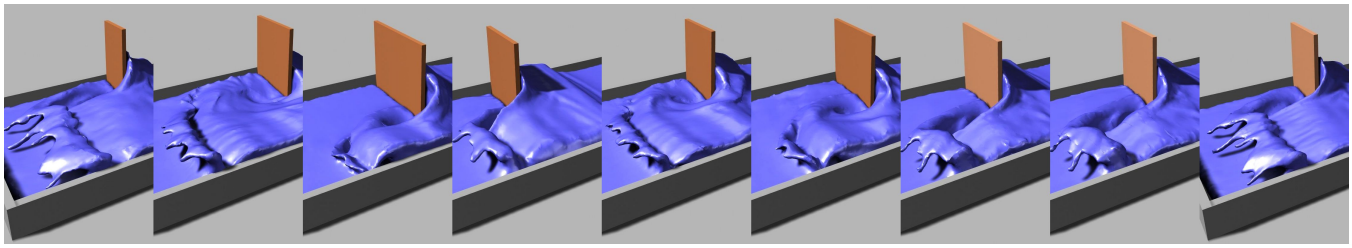

Figure 1: *Our method can quickly generate an entire family of fluid simulations from a small set of inputs. Here, we generate a large set of animations of liquid colliding with walls of varying shapes and locations.*

Abstract

We present a method for smoothly blending between existing liquid animations. We introduce a semi-automatic method for matching two existing liquid animations, which we use to create new fluid motion that plausibly interpolates the input. Our contributions include a new space-time non-rigid iterative closest point algorithm that incorporates user guidance, a subsampling technique for efficient registration of meshes with millions of vertices, and a fast surface extraction algorithm that produces 3D triangle meshes from a 4D space-time surface. Our technique can be used to instantly create hundreds of new simulations, or to interactively explore complex parameter spaces. Our method is guaranteed to produce output that does not deviate from the input animations, and it generalizes to multiple dimensions. Because our method runs at interactive rates after the initial precomputation step, it has potential applications in games and training simulations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: Fluid simulation, non-rigid registration, shape blending.

Links: DL PDF WEB VIDEO

1 Introduction

The ability to direct and fine-tune visual effects is one of the main reasons for their popularity. This is especially true for effects that are as difficult to control as fluids. Because modern fluid simulators expose numerous important parameters to the artist, it involves a certain degree of trial and error to produce a fluid animation with a specific behavior.

A common approach in special effects production is to run simulations in batches to explore the parameter space. Afterward, the artist and supervisor can select the simulation that best matches the desired goal. If none of the outputs is chosen, the parameter range is narrowed down to run additional simulations. However, there are several downsides to this approach: the new simulations can result in undesired behavior (like a distracting splash in the center of attention), it may require an excessive number of iterations of expensive fluid simulations, and the data from early simulations are wastefully discarded.

We propose to solve this problem by smoothly blending between existing fluid animations. We first develop a semi-automatic method for matching two existing liquid animations. Once we have this matching, we are able to immediately create new fluid motion that plausibly interpolates the input.

Our method allows us to instantly synthesize hundreds of new animations from a sparse set of input samples and can be used to interactively explore the parameter space. Further, since our technique is based on interpolation, it guarantees that the output will not deviate significantly from the provided inputs. After the initial precomputation phase, new animations can be generated on the fly at interactive rates. As a consequence, it has potential applications in games or training simulators where it can be used in place of a real-time fluid simulation.

The key contributions of our work are as follows:

- A new approach to interpolate between free surface fluid simulations with changing topologies.
- A 4D non-rigid iterative closest point algorithm that incorporates user guidance.
- A subsampling technique for efficient registration of meshes with millions of vertices.
- A fast surface extraction algorithm that produces 3D triangle meshes from a 4D space-time surface.

*e-mail: kraveendran@gatech.edu
†e-mail: wojtan@ist.ac.at
‡email: nils@thuerey.de
§e-mail: turk@cc.gatech.edu

2 Problem

We aim to interpolate between two or more existing liquid animations. A liquid animation consists of a set of closed manifold triangle meshes, with each mesh representing the liquid surface at a specified time. Each individual surface mesh may have an arbitrary number of voids, connected components, and tunnels (representing bubbles, droplets, and tumbling waves).

Naively matching two such mesh sequences together frame-by-frame is problematic, because each pair of meshes may have significantly different topologies, and because the surfaces must be temporally coherent as they evolve over time. The topology problems can in principle be solved by blending implicit surfaces [Cohen-Or et al. 1998] or more sophisticated mesh-based alignment techniques [Bojsen-Hansen et al. 2012], but they cannot guarantee temporal coherence without additional work.

For this reason, we opt not to match together individual animation frames, but propose to match all of the frames at once. We do this by concatenating all of the triangle meshes together into a 3D hypersurface in 4D space-time (Section 4) and then performing a high-dimensional non-rigid registration (Section 5). We wish to deform one space-time surface to match another one while keeping the resulting deformation smooth. This strategy amounts to minimizing both a "fitting" energy and a "smoothness" energy (Section 6).

Once we have successfully aligned a few space-time surfaces, we can interpolate between them to produce intermediate motion (Section 7). Finally, we can rapidly extract individual frames from the space-time surface to produce the output animations (Section 8). Figure 2 illustrates the steps in our method, using 2D animations as input so that we can visualize the entire space-time surface.

3 Related work

Although our algorithm theoretically applies to a broad class of surfaces that deform strongly over time, our primary focuses is on liquid animations. Fluid simulation has become an established field of computer graphics research, after the seminal work of Foster and Metaxas [1996] and the popularization of stable advection routines [Stam 1999]. In addition to such Eulerian approaches, the state-of-the-art is steadily advanced by purely particle-based [Müller et al. 2003; Solenthaler and Pajarola 2009] and hybrid approaches [Zhu and Bridson 2005].

As our method takes surface data as input, it is oblivious to the particular algorithm employed for solving the Navier-Stokes equations. However, the choice of a surface tracker has implications on what kind surface data is generated by the simulation. A popular class of surface tracking algorithms is based on level-sets [Osher and Fedkiw 2002], and particle level-sets [Enright et al. 2003]. Another class of methods uses an explicit surface representation [Brochu et al. 2010; Wojtan et al. 2010; Misztal et al. 2012]. These methods better preserve fine details, but they require additional work to retain well-shaped elements and handle changes in topology. For particle based methods a variety of surface reconstruction approaches have been proposed, e.g., using anisotropic kernels [Yu and Turk 2010]. While we use a surface tracker as described by Wojtan et al. [2010], our method could be similarly applied to any other surface tracker, as long as a triangle mesh is generated for each frame of the animation.

Several previous works share our goal of modifying an existing fluid simulation, using non-linear optimization [McNamara et al. 2004] or forces [Shi and Yu 2005; Thuerey et al. 2006] to achieve this goal. Similarly, some researchers have used guide behaviors to make liquid surfaces follow a predefined path or shape [Nielsen and Bridson 2011; Raveendran et al. 2012; Pan et al. 2013]. However, in contrast to previous work, our goal is to very efficiently generate in-betweens for a pre-computed batch of fluid simulations — a work-flow that has not been targeted by any previous method.

There is a rich literature of work relating to the registration of surfaces and point clouds in both computer vision and computer graphics. The iterated closest point method (ICP) was introduced by Besl and McKay [1992] and numerous improvements have been suggested over the years [Hähnel et al. 2003; Gelfand et al. 2003; Rusinkiewicz and Levoy 2001; Brown and Rusinkiewicz 2007]. Our work is closely related to the non-rigid ICP algorithms [Amberg et al. 2007; Li et al. 2009]. This technique has been applied to problems such as completion of dynamic shapes [Li et al. 2012] where they reconstruct a temporally coherent and water-tight sequence of meshes from data captured by multi-view acquisition systems. Non-rigid ICP has also been applied to the tracking of shapes with changing topology such as liquid surfaces [Bojsen-Hansen et al. 2012]. Unlike these methods which register a pair of 3D meshes, in this paper, we extend the non-rigid ICP algorithm to register entire animations of liquids. This spacetime formulation lets us handle a number of scenarios that would be difficult or impossible for frame-by-frame registration algorithms. For instance, the water drop example requires retiming; otherwise a floating drop and pool (two components) would be awkwardly forced to align with splash geometry (one component).

During each iteration of the ICP algorithm, we deform the source animation to move all vertices towards their current correspondences. We perform this deformation using a variation of the embedded deformation model first introduced by Sumner and colleagues in [Sumner et al. 2007] and extended by Li and colleagues in [Li et al. 2009]. Our deformations are applied in 4D spacetime and we do not impose any non-linear constraints on the columns of the affine transformation matrix at each node. This leads to a simple weighted linear squares solve that can be solved using standard methods.

Aside from registration with ICP, researchers have explored a variety of other approaches for matching surfaces. Szeliski et al. proposed to use splines [Szeliski 1996], while others used local similarity transforms [Papazov and Burschka 2011], intrinsic features [Gelfand et al. 2005], and high-order graph matching [Zeng et al. 2010]. Numerous methods in computer graphics morph between surfaces of different topology using an implicit representation [Cohen-Or et al. 1998; Breen and Whitaker 2001; Turk and O'Brien 1999]. These methods are convenient because they do not require explicit correspondence generation or advanced meshing techniques. On the other hand, they are unable to use correspondences to align salient features. For this reason, we opt to use ICP to handle the majority of the registration, but our approach could be combined with methods such as those using implicit representations for registering additional small-scale features.

The idea of treating animation data as a surface or volume in a higher dimensional space has been used by graphics researchers for various purposes. For instance, Klein and colleagues applied this notion to frames of a video sequence and created a rendering solid that could be used to render stylized videos [Klein et al. 2002]. Kwatra and Rossignac applied compression algorithms to the bounding triangles of a space-time volume of 2D cel animation data [Kwatra and Rossignac 2002]. Finally, Schmid and colleagues construct a time aggregate object by connecting triangle meshes in time and use this data structure to produce stylized blurring and stroboscopic images [Schmid et al. 2010]. Similarly, our algorithm creates a spacetime mesh per animation by linking up frames of a liquid surface. However, we also need to consider changes in con-

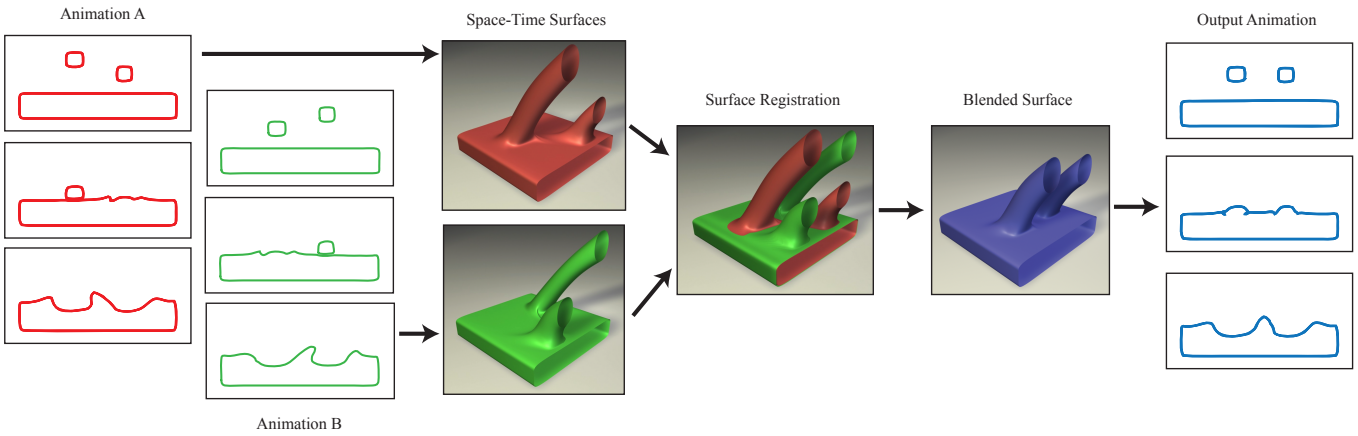

Figure 2: *Overview of our method, using 2D animations for illustrative purposes. The input animations show two drops falling into a pool. The space-time meshes are 3D surface that are registered to one another. A blended space-time mesh is then created, and this is sliced to produce final animation frames.*

nectivity and topology between frames due to the evolving fluid surface and we handle these with tetrahedra instead of bilinear patches.

4 Space-time surfaces

Our method takes two animations as input: a source A and a target B. Each animation is a sequence of triangulated meshes (corresponding to a fluid surface evolving over time) along with per-vertex velocities at each frame. We convert each animation into a space-time mesh, which is a higher dimensional object obtained by linking up vertices of successive frames of the animation in time (by using the velocity field). There are several reasons for choosing this particular representation:

- We can treat each animation as a manifold with boundary and hence as continuous data.
- The motion which results from deforming an animation will be temporally coherent due to this continuity.
- Changes in topology such as merges or splits do not need special treatment since the space-time surface remains a single connected component.
- It is a sparse/compact representation compared to others that use volumetric data.

4.1 Construction

To construct a space-time surface, we first represent each vertex $\boldsymbol{v}$ at time t as $(v_x, v_y, v_z, t{\cdot}e) \in \mathbb{R}^4$ where e is the average length of all edges in the first frame of the animation sequence. This ensures that space and time dimensions are scaled uniformly. We then wish to connect the individual meshes in time. Instead of using a generic animation reconstruction algorithm [Tevs et al. 2012], we can take advantage of the information embedded in our fluid simulations. We advect each vertex $\boldsymbol{v}_i^t$ using the velocity field $\boldsymbol{u}_i^t$ for an interval Δt to its predicted position $\widetilde{\boldsymbol{v}}_i^t$. We then find the vertex $\boldsymbol{v}_j^{t+1}$ on frame $t{+}1$ that is closest to $\widetilde{\boldsymbol{v}}_i^t$. If the geodesic distance between the two is less than $2e$, then we create a link between $\boldsymbol{v}_i^t$ and $\boldsymbol{v}_j^{t+1}$. This link will not exist when a vertex is deleted during a topology change. For instance, two surfaces will collide and disappear where they merge together; a vertex on one of these colliding surfaces will have no corresponding vertex in the next frame.

With this time link information, we can compute normals by creating a vector that is orthogonal to the tangents in space and in time; we now have an oriented surface corresponding to an animation. Note that this representation is not a fully tessellated mesh in space-time, but instead is a graph where each vertex has a set of neighbors in space and up to two neighbors in time. In other words, this graph provides connectivity information as well as orientation at each vertex by stacking frames next to each other in time and loosely connecting them with links that are guided by the velocity field. When required, we can construct a local approximation of the space-time surface near each vertex on the fly as described in Section 5.2.

5 Registering space-time surfaces

The registration of two fluid animations is a challenging problem for a couple of reasons. Firstly, it is highly unlikely that the space-time mesh corresponding to the source can be mapped onto that of the target through a single rigid transformation. Secondly, due to the non-linear nature of Navier Stokes equations, the surface can stretch or compress to a large degree. As a result, even the assumption of local rigidity is far too constraining. This also means that using an intrinsic characteristic such as Gauss curvature to automatically create correspondences between the two surfaces is not as robust as it normally would be for aligning meshes with moderate deformations.

To solve this problem, we use a non-rigid iterated closest point algorithm (non-rigid ICP). The input to this algorithm consists of two space-time surfaces A and B. The output is a set of correspondences (one for each vertex of A) that lie on the surface B. The basic idea behind is ICP is to deform the source mesh using a small number of deformation nodes such that its vertices end up at the points nearest to them (correspondences) on the target. Then, new correspondences are computed for this deformed mesh and the process is repeated until convergence.

When applied to two space-time meshes, non-rigid ICP moves vertices of the source mesh in both space and time to best match the target. This enables us to align two animations that have seemingly contradictory constraints such as in Figure 4. This example features

impact events that occur in a distinctly different order between the two input animations. Our algorithm deforms the two animations in space, and more importantly time, to create correspondences between the two impacts. This allows us to generate an animation where both drops hit the surface simultaneously.

5.1 Local deformation model

In order to deform one animation into another, we use a collection of local surface deformations. We uniformly sample the space-time surface with deformation nodes that are placed at a subset of the vertices. Each node has an affine transform attached to it (which is split into a 4×4 matrix $\boldsymbol{A}$ and a 4×1 translation vector $\boldsymbol{b}$) and influences all vertices within a given radius (measured using geodesic distance computed by a fast marching algorithm). Hence, we have to solve for 20 unknowns per node such that the resulting surface is as close to the target as possible. The position $\boldsymbol{v_i^{k+1}}$ of a vertex at iteration $k+1$ is determined by the weighted sum of all nodes $\boldsymbol{n_j}$ that influence it:

$$\boldsymbol{v_i^{k+1}} = \sum_j w\left(\boldsymbol{n_j}, \boldsymbol{v_i^A}\right)\left(\boldsymbol{A_j}\left(\boldsymbol{v_i^k} - \boldsymbol{n_j}\right) + \boldsymbol{n_j} + \boldsymbol{b_j}\right) \quad (1)$$

where w is a weight computed based on the geodesic distance between $\boldsymbol{n_j}$ and $\boldsymbol{v_i^A}$ (i.e. the distance between the node and the vertex on the undeformed mesh).

5.2 Finding correspondences

The ICP algorithm relies on finding good correspondences at the beginning of each iteration. Given a vertex on the deformed version of the source mesh, we set its corresponding vertex to the closest point on the target whose normals point in the same direction. The user can also prescribe a sparse set of correspondences as described in Section 6.

Since we do not have an explicit global tessellation of the space-time surface, we first find the closest vertex on the target (using a kd-tree lookup) and then find the closest point by projecting onto the tetrahedra that surround it. To accomplish this, we construct the local surface near the vertex by creating triangular prisms for each face incident at that vertex. We can build these prisms by looking up the forward neighbor in time $(t+1)$ for each vertex of a triangular face at time t. Next, we split up each prism into three tetrahedra and find the closest point on these tetrahedra with a compatible normal.

In our experience, using the closest point rather than the closest vertex makes the registration more robust and is worth the additional computational expense.

5.3 Handling thin sheets and droplets

Splashing liquids often exhibit thin sheets and flying droplets (see Figure 3). These features tend to evolve very differently for even minor perturbations. In many cases, there may not be an obvious mapping between these features for two simulations. Further, if a corresponding feature does not exist in the target, it might get deformed incorrectly onto another region of the target mesh and might lead to unrealistic behaviour in the interpolated output. As a result, if the user does not explicitly provide a correspondence for a thin sheet or a droplet, we do not attempt to automatically register them onto the target.

To produce plausible interpolations, we preserve such features from the source animation instead of deforming them into an incorrect portion on the target. The easiest way to do this is to exclude from the fitting energy functions all vertices that are a part of a thin sheet or a droplet. Note that these vertices are still influenced by the nearby deformation nodes, and the smoothness energy will ensure that their behavior is consistent with the global deformation.

We perform a connected component search of the triangle mesh at each frame. Any component with fewer than 150 vertices is considered to be a droplet and all of its vertices are excluded from the solve. We preserve these drops without distorting them by finding their centroids, moving these centroids through the displacement field (computed using Equation 1) and finally reconstructing the vertices.Next, we find vertices that are part of a thin sheet. For each vertex, we query the kd-tree for the 50 nearest neighbors within a radius equal to the minimum sheet thickness (typically the width of a grid cell). If more than 10% of its nearby points have a normal that is opposite to its own, we flag all vertices as belonging to a thin sheet. Further, we grow out this region by two rings (i.e. all neighbouring vertices as well as their neighbours) to remove any stray vertices that might not have been identified.

6 User-guided registration

A drawback of the closest point search in Section 5.2 is that it may not always find a corresponding point that we might want. This is of particular concern because a set of poor initial correspondences can easily cause non-rigid ICP to settle on a local minimum. For instance, in Figure 4, the vertices on the top of the lower of the drops will pick their corresponding points on the top of the pool instead of those on the drop in B because of their proximity. To resolve this ambiguity, the user can provide a sparse set of correspondences between pairs of points on A and B (for instance, for the top and bottom points of the drop). Note that correspondences are not restricted to points that lie on the same frame (i.e. same time) in both animations and are instead specified in space-time. As a result, we can map events that occur at different points in time to each other (such as the impacts of two droplets).

We specify user correspondences using a simple UI in Maya [Autodesk 2013]. For a demonstration of our interface, please view the supplementary video. To create a correspondence, the user first selects the type of correspondence and then selects two vertices (one on the source and the other on the target). To simplify the task, we allow three kinds of correspondences:

1. Point : The most basic form that maps one point in space-time onto another.

2. Trajectory : This is used when one spatial feature needs to be mapped onto another for a duration of time. For instance, if one wants to specify that the tips of the splashes need to be aligned, then a trajectory correspondence will ensure that the alignment persists until the splash dies out. If the user does not specify the length of the trajectory, the algorithm will try to preserve this correspondence for the entire animation. Note that we require only the starting points because we can use the velocity field to trace the correspondence through time.

3. Space : This pins a vertex to a point in space, but allows it to slide in time. It is useful if the two simulations have different solid boundary conditions.

Using these three types of correspondences, the user can quickly map salient features from one simulation onto the other. Further, the user can specify whether the correspondence needs to be strictly enforced (hard) or not (soft). In general, a hard correspondence will significantly influence the output of the ICP algorithm, while a soft correspondence may be ignored if the rest of the mesh disagrees with that particular choice. On average, the user only needs to specify 10 to 20 correspondences in total for an animation consisting of

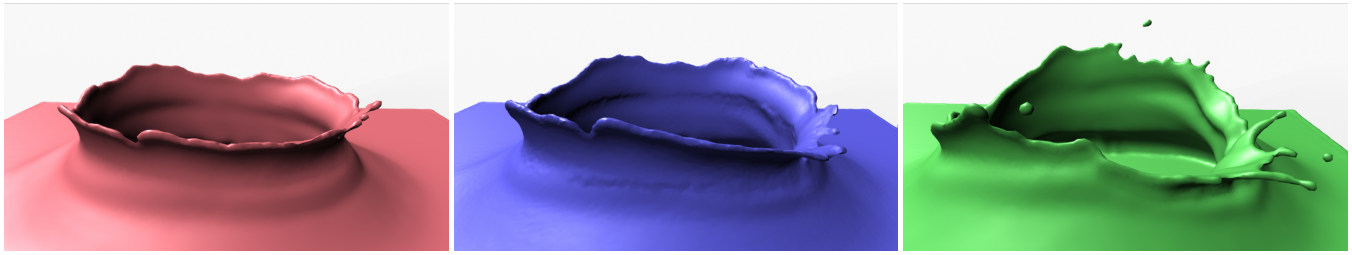

Figure 3: *The* 50 *percent interpolation (middle) of two crown splash animations (left and right).*

10 million space-time vertices.

We create deformation nodes at each vertex of the source animation that is specified as a user correspondence. These correspondences affect the registration in a couple of ways. First, they are used in an initial diffusion step where the user correspondences are propagated to other deformation nodes. We fix the translation vector $\boldsymbol{b}$ for all user correspondences, and then run passes of diffusion using the vector valued heat equation on the translation vectors. This helps to provide a globally consistent initialization for the deformed mesh. Secondly, user correspondences override the closest point search for that vertex. Soft correspondences are only applied during the diffusion step and the closest point search and do not influence other deformation nodes directly. In contrast, hard correspondences serve as equality constraints on one or more of the elements (x, y, z, t) of the translation vector $\boldsymbol{b}$ for the deformation node during the solve. This means that nearby vertices as well as deformation nodes are affected by a hard correspondence.

We implement the constraints for the hard correspondences in the following way:

1. Point: We completely specify all elements of the translation vector for the node.
2. Trajectory: We trace each point through the velocity field and create deformation nodes at uniform intervals along time (typically every 10 frames). The translation vectors of these deformation nodes are fully specified. In other words, a trajectory correspondence is a set of point correspondences for a single vertex over time.
3. Space: We create deformation nodes at the specified spatial location at uniform intervals along time and fix (x, y, z) of the translation vector while allowing it to move along time.

During the solve, all specified elements are removed as free variables and their known values are substituted into the energy functions and Jacobian.

6.1 Energy functions

One of the characteristics of our fluid animations is that they have a large degree of local deformation, but are already globally aligned (in that no rotation or translation is required to put them both in the same coordinate frame). We have found that an energy term enforcing rigidity, which is used in many other ICP algorithms, causes significantly worse registrations for the strongly deforming meshes we are dealing with. As a consequence, we employ only fitting and smoothness energies, which will be described in more detail below.

Fitting The fitting energy functions measure how close the currently deformed version of the source is to the target. For each point $\boldsymbol{v_i}$ on the source mesh that has a corresponding point $\boldsymbol{c_i}$ on the target, we have a point-to-point energy that computes the Euclidean distance separating them. Similarly, we also have a point-to-plane energy that permits sliding along the plane of the corresponding point and helps to smooth out the energy landscape.

$$E_{\text{point}} = \sum_i \|\boldsymbol{v_i^{k+1}} - \boldsymbol{c_i}\|_2^2 \tag{2}$$

$$E_{\text{plane}} = \sum_i |\langle \boldsymbol{N_i},\ \boldsymbol{v_i^{k+1}} - \boldsymbol{c_i} \rangle|^2 \tag{3}$$

Smoothness The smoothness energy term ensures that affine transforms of adjacent deformation nodes are similar to each other. For each node, this energy measures the distance between its predicted position using the affine transform of its neighbor and its actual position based on its translation vector. The energy is defined as follows:

$$E_s = \sum_{\boldsymbol{n_i}} \sum_{\boldsymbol{n_j}} w(\boldsymbol{n_i}, \boldsymbol{n_j}) \|\boldsymbol{A_i}\,(\boldsymbol{n_j} - \boldsymbol{n_i}) + \boldsymbol{n_i} + \boldsymbol{b_i} - (\boldsymbol{n_j} + \boldsymbol{b_j})\,\|_2^2 \tag{4}$$

6.2 Subsampling

Each fitting energy function for a vertex adds 4 rows to the Jacobian. Note that each deformation node has compact support, so, on average each vertex in 4D spacetime is influenced by 27 nodes. The space-time meshes for a 5 second animation clip typically contain several million vertices, and as a result, if we used the entire mesh for registration, we would end up with a Jacobian matrix that has on the order of tens of millions of rows. Even with sparse matrices, such large sizes place unreasonable requirements on memory and add to the computational cost of the algorithm.

Instead, we subsample the mesh by randomly picking 10% of the vertices for each iteration of the algorithm. We retain 20% of the sampled vertices from the previous iteration by selecting those with the highest error. This random subsampling scheme speeds up the algorithm by close to an order of magnitude, reduces the memory footprint, and works flawlessly in practice.

6.3 Solving the linear system

We define the total energy of the system as a weighted sum of the energies defined above:

$$E_{\text{total}} = \gamma_s E_s + \gamma_{\text{point}} E_{\text{point}} + \gamma_{\text{plane}} E_{\text{plane}} \tag{5}$$

where the weights γ_s, γ_{point}, and γ_{plane} are set to 250, 0.1, and 1 in our implementation, respectively.

Since each term in E_{total} is at most quadratic in our free variables (the entries of the affine transformation matrix $\boldsymbol{A_i}$ and translation

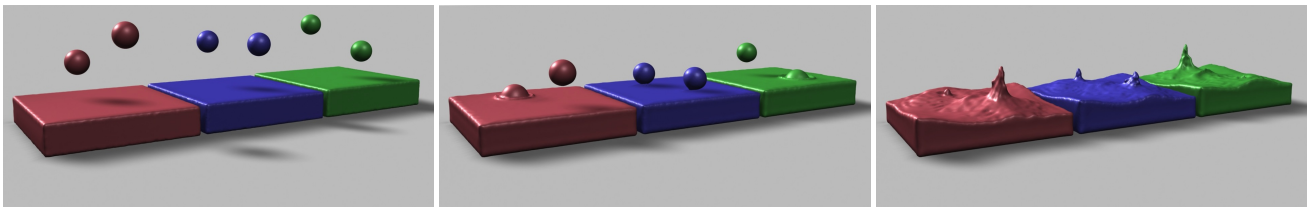

Figure 4: *Animations of two spheres of water that are dropped into a pool. The red and green surfaces are from the input animations. Notice that in each, the spheres strike the surface at different times. The blue images show our blending result, which causes both spheres to strike the pool at the same time.*

vector $\boldsymbol{b_i}$) , we can view it as a weighted least squares problem. We solve this using normal equations:

$$\left(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} + \lambda\boldsymbol{I}\right)\boldsymbol{\Delta x} = -\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{E} \quad (6)$$

$$\boldsymbol{x}^{\boldsymbol{k+1}} = \boldsymbol{x}^{\boldsymbol{k}} + \boldsymbol{\Delta x} \quad (7)$$

where $\boldsymbol{J}$ is the Jacobian of the system, $\boldsymbol{W}$ is a diagonal weight matrix, $\boldsymbol{E}$ is the column vector created from each squared term in E_{total}, λ is a damping factor (typically set to 1e-6) and $\boldsymbol{x}$ is a column vector created by concatenating all free variables from each deformation node.

The left hand side of the system is a dense, symmetric positive definite matrix. Note that regardless of the number of vertices in the mesh, the size of the resulting linear system depends only on the number of deformation nodes. For relatively small matrices, we use a Cholesky decomposition followed by backsubstition to solve the system. For large matrices ($> 10,000$ rows), we employ a preconditioned conjugate gradient solver. After solving the system, we deform the source mesh using the new affine transforms and then recompute correspondences. We repeat this for a number of iterations (usually $10-20$) or until the difference between the total energy in successive iterations falls below a threshold. Our system also allows for an incremental or iterative registration process where the user can add new correspondences to fine-tune a previous registration. To incorporate the newly added correspondence, we create a new deformation node on the source mesh at that position in space-time. Next, we recompute the node and vertex weights for all points near newly added correspondence. We then load the previously computed values for all of the free variables and apply a few passes of diffusion on the translation vectors to ensure that the solution is smooth and resume the registration process.

7 Interpolation

Having registered the two animations, we now have a correspondence on B for each vertex of A. Given an interpolation weight α, we can produce an intermediate space-time mesh by linearly interpolating between the positions of vertices of A and their corresponding points on B. A given vertex $\boldsymbol{v_i^A}$ in mesh A will have an interpolated position $\boldsymbol{v_i^I} = (1-\alpha)\boldsymbol{v_i^A} + \alpha\boldsymbol{v_i^B}$. Note that the interpolated mesh has the same connectivity as the space-time mesh of A because of the way the correspondences were created.

When blending between two animations, it is possible to set the blend weight α to be outside the range of zero to one. Blend weights outside this range correspond to extrapolations. We have found that for modest numerical values (e.g. $\alpha = 1.25$), such extrapolations produce plausible animation results.

We can also use our blending approach to create novel animations between three or more input animations. Assume that we have three animations, A, B, and C, and that we have a correspondence between an animation A and B, and another correspondence between A and C. Given barycentric blending weights α, β, we can produce new vertex positions according to

$$\boldsymbol{v_i^I} = (1-\alpha-\beta)\boldsymbol{v_i^A} + \alpha\boldsymbol{v_i^B} + \beta\boldsymbol{v_i^C}. \quad (8)$$

Using this technique, our example animations can span more than just a single parameter family of animations. We can think of each example animation as a sample in a multi-dimensional parameter space. We can produce intermediate animations anywhere within the convex hull of these parameter space samples by performing barycentric blending between the three nearest surrounding samples.

Algorithm 1 Registering liquid animations

Require: Two sequences of meshes with per-vertex velocities A, B, user correspondences u
$A_{\text{st}}, B_{\text{st}} \leftarrow$ Construct space-time meshes from A and B.
$nodes \leftarrow$ Create deformation nodes on A_{st}.
Precompute deformation node weights for each vertex of A_{st}.
Diffuse user correspondences u to nearby deformation nodes.
while $k < maxIterations$ **do**
  $A_{defo} \leftarrow$ Deform A_{st} by using $nodes$.
  $\boldsymbol{v} \leftarrow$ Randomly subsample m vertices of A_{defo}.
  $\boldsymbol{c} \leftarrow$ Compute closest points on B_{st} for each $\boldsymbol{v}$.
  $E \leftarrow$ Compute energy vector using correspondences $\boldsymbol{c}$.
  $nodes \leftarrow$ New affine transformations from minimizing E.
  $k \leftarrow k + 1$
end while
Compute final registered mesh R from $nodes$.
return R

8 Surface Extraction

After registering two space-time surfaces and forming a new surface through interpolation, we need to extract per-frame triangle meshes for rendering. Note that because our registration allows motion in both space and time, the vertices from a single frame (that used to all lie in a common plane in time) will no longer have the same time values. We cannot just use these vertices as our meshes to render. We extract *new* triangle surfaces from an interpolated space-time mesh by first constructing an explicit tetrahedralization of the entire space-time mesh. The purpose of this tetrahedralization is to "fill in" the regions between pairs of meshes that come from the different discrete frame times. We connect each triangle with its corresponding triangle in the next frame (using tetrahedra) to create the full tetrahedralization. We then slice this tetrahedral mesh with hyperplanes of constant time, and each such slice yields a triangle mesh for a given frame time.

During registration between space-time meshes, we only use a

Algorithm 2 Interpolating liquid animations

Require: Mesh A_{st}, registration R, interpolation weight α, time t
1: $A_{interpol} \leftarrow (1-\alpha)A_{st} + \alpha \cdot R$
2: $triangles \leftarrow$ Intersect $A_{interpol}$ with hyperplane at t.
3: **return** $triangles$

loose correspondence between adjacent frames in an animation, through time links. For surface extraction, however, we need an explicit tetrahedralization. We begin forming these tetrahedra by using the time links to help us define triangular prisms. Each triangle from one frame has three links to another triangle on the next frame, and this defines a prism. We then split up each such prism into three tetrahedra as shown in the Appendix (Figure 9, left). We take care to ensure that adjacent triangular prisms have a consistent tetrahedralization, otherwise the resulting surface could have holes or overlaps. More specifically, it is important that the coinciding quadrilateral faces of adjacent prisms make use of the same diagonal to divide the quad into two triangles. In the Appendix, we prove that it is possible to guarantee these matching diagonals for a manifold triangular mesh.

Because our surface tracker attempts to maintain the surface triangulation between frames of the animation, almost every triangle from one frame has a matching triangle in the next. In these cases, the corresponding prisms are well defined. In the cases where our tracker performs mesh clean-up operations such as edge splits, edge collapses, or edge swaps, we can still create a small group of tetrahedra to fill between the time slices. During topology changes, however, the mesh is locally re-built, and forming the tetrahedra is problematic. In such cases, we have left holes in the tetrahedral mesh, and this results in small holes in the extracted surfaces. We currently fill these holes by introducing a new vertex at the centroid of the hole vertices and constructing a triangle fan that radiates from the centroid to all of the boundary edges.

After slicing the tetrahedral mesh by a hyperplane, the resulting mesh contains many sliver triangles. These sliver triangles cause noticeable shading discontinuities if they are retained. We perform one step of Laplacian smoothing [Taubin 1995] to even out the triangle shapes, and the resulting mesh looks considerably improved during final rendering.

Our space-time surface registration routine is summarized in Algorithm 1, and the interpolation and surface extraction are reviewed in Algorithm 2.

9 Results

All of our examples were run on a workstation with an Intel Xenon ES processor with six cores that runs at 3.2 GHz and has 72 Gbytes of main memory. The fluid simulation code and our registration code are written in C++, and they are both multithreaded. For our matrix solves, we use the Intel MKL library and Eigen. Figure 6 gives simulation, registration, and mesh extraction times for each of our examples. Note that the entire registration process requires less time than the time it takes to run an individual fluid simulation. Also, once the registration has been performed, new animation meshes can be produced for any blend weights in a fraction of the time it takes to perform either the simulation or registration.

Accompanying this paper is a video that shows our animation results. We use a common color scheme in all of our examples (still images and video). Red and green meshes indicate original animation sequences that were generated by running a standard fluid simulation. Blue meshes indicate blended results that were created using our method. During registration, we always deform the red space-time mesh to match that of the green.

One of the strengths of our approach is that it can alter the timing of events in an animation. An extreme example of this is demonstrated in Figure 4, where two drops of water are released and strike a pool. In one animation (red) the leftmost drop hits the water first, and in another animation (green), the rightmost drop hits first. Using our animation blending approach, we can create an entire family of animations in which the two drops hit the water at various times. For instance, the blue animation shows a variant in which the two drops strike the water simultaneously. Note that this result would not be possible if we deformed our animations purely in space and not in time.

Our dam break example shows that we can interpolate between more than two animations (Figure 8). In this set of animations, a block of water is released at one side of a long pool, and this forms a wave that sweeps the length of the pool. A wall blocks a portion of the pool, so the water must sweep around this wall. We simulated animations for each of four different positions and widths for this wall, giving us a two-dimensional family of parameters (wall width, wall position). These two distinct parameters allow us to create new animations anywhere within this two dimensional parameter space. In the figure, we show stills from nine different animations to show the range of variations that this allows.

The crown splash example (see Figure 3) demonstrates how our algorithm works on input that has droplets, thin sheets and numerous topology changes. Despite the relative coarse sampling of the mesh, the interpolated result captures the global behaviour of the two crowns. However, the droplets in the interpolation depend on the choice of the source animation.

We compared our interpolated results to actual simulations that were run with appropriate parameter values. The two animations are qualitatively similar, however there are certain differences as can be seen in Figure 5. Please see the accompanying video for more examples.

Figure 7 shows a fluid duck being thrown into a pool of water. The original animations have the ducks thrown at few different angles,

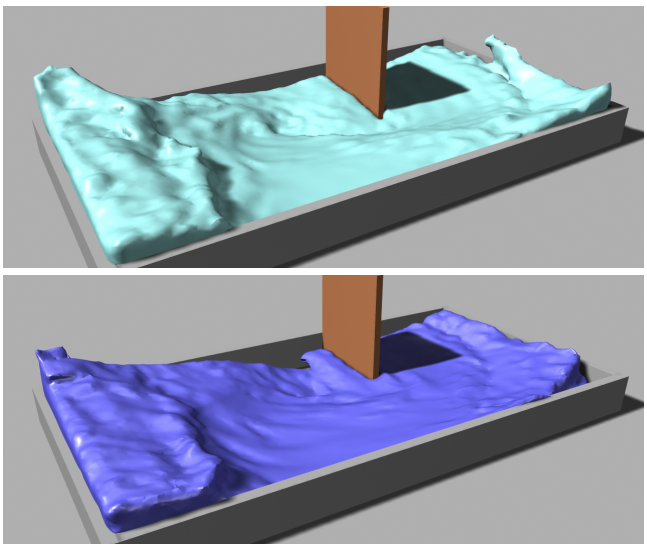

Figure 5: *A comparison of the simulated result (top, in cyan) against our interpolated result (bottom, in blue) for the dambreak scenario with the wall width set to* 50 *percent .*

Animation	Number of Correspond.	Simulation Time	Regist. Time	Extract Time
Two Drops	8	35.2	14.8	0.31
Dam Break	20	40.7	14.1	0.25
Duck	10	22.1	15.7	0.35
Crown Splash	20	67	22	1.9

Figure 6: *This table gives the average number of correspondences per example, simulation time, registration time, and the time it takes to extract all meshes in the entire sequence. All sequences consist of* 150 *frames except for the crown splash (*75 *frames). All times are given in minutes, so we extract about 5 meshes per second for input meshes with* 50*k vertices.*

causing them to splash into the pool at different locations. We can interpolate between any pair of input animations, allowing us to span a continuous range of possible throwing angles. In addition, we aimed two of the ducks at the wall, which adds a discontinuity to the behavior of the animation. If we select one of our input animations to capture this discontinuity, we can smoothly interpolate across this event. Interpolating between the two ducks that hit the wall allows us to produce duck strikes along a range of positions on the wall. We can even use extreme blending weights ($\alpha = 1.25$) to produce new animations in which the duck hits the wall above the highest example animation.

10 Discussion and Limitations

Currently, our method has several limitations that suggest possible directions for future research. First, our technique is aimed at interpolating between simulations that are qualitatively similar and that do not have highly divergent behaviors. This means that we might need additional simulation samples for parameters that produce highly discontinuous behavior. For instance, our method is unlikely to produce plausible behavior near solid boundaries if none of the samples involved the solid boundary. It would be useful if we could automatically determine if two simulations are too far apart by using the energy functions.

Second, we rely on the user to identify a few salient features in each simulation. In our experience, these features coincide with regions of high curvature in space or in time. One possibility is to build up a statistical model of such features by learning from user correspondences, and then automate the process. At the very least, we could further reduce the effort required from the users by only involving them in a simple verification step.

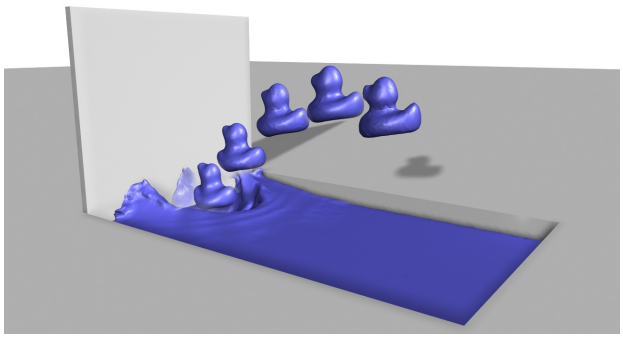

Figure 7: *This composite image shows a duck being thrown into a pool of water. This animation was produced by blending between two animations of ducks that were thrown at different angles. Please watch the video to see the continuous variation of duck trajectories that we can span using blending.*

Another issue that we ran into was the memory consumption of the algorithm. Even with subsampling and a relatively coarse sampling of deformation nodes, registering two detailed animations requires between 30 to 50 GB of RAM. At the same time, the subsampling technique indicates that a multi-resolution approach to this problem is likely to work. It might be possible to run a coarse low resolution ICP and then break up the space-time mesh into smaller pieces with higher sampling densities of both vertices and deformation nodes. Another implication of the coarse sampling is that small scale details of the source animation tend to be preserved in the interpolated result. The animator has a choice of designating either animation as the source (please the crown splash video for an example of this).

We should also note that our method can still run into the problem of local minima that is associated with non-rigid ICP. However, this has not been problematic in our tests as undesirable results can be prevented with correspondences provided by the user.

Finally, our surface extraction algorithm produces meshes that can change their triangulation fairly often. This may sometimes result in flickering especially at lower resolutions and can be ameliorated by a pass of temporal smoothing.

11 Conclusion

This paper introduces a flexible new method for the generation and control of liquid animations. We can instantly generate new fluid animations on the fly by simply interpolating between existing simulations, and we can plausibly fine-tune a simulation by warping its space-time representation. In the future, we envision a fully automated version of our system that generates an infinite set of simulations from a given range of input parameters. We believe a system like this would make a significant impact on the state-of-the-art in special effects production, and it could easily generalize to other phenomena beyond liquids.

12 Acknowledgements

This work was funded by NSF grant IIS-1130934. We thank Hao Li for the useful discussions about non-rigid ICP, Mark Luffel for his help with the hole-filling algorithm and David Hahn for the volume preserving edge collapse code. We are grateful to our anonymous reviewers for their suggestions and feedback.

References

AMBERG, B., ROMDHANI, S., AND VETTER, T. 2007. Optimal step nonrigid ICP algorithms for surface registration. In *IEEE Conf. Computer Vision and Pattern Recognition, CVPR '07*, 1–8.

AUTODESK, 2013. Maya software.

BESL, P. J., AND MCKAY, N. D. 1992. A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell. 14*, 2 (Feb.), 239–256.

BOJSEN-HANSEN, M., LI, H., AND WOJTAN, C. 2012. Tracking surfaces with evolving topology. *ACM Trans. Graph. 31*, 4 (July), 53:1–53:10.

BREEN, D. E., AND WHITAKER, R. T. 2001. A level-set approach for the metamorphosis of solid models. *IEEE Trans. Visualization and Computer Graphics 7*, 2 (Apr.), 173–192.

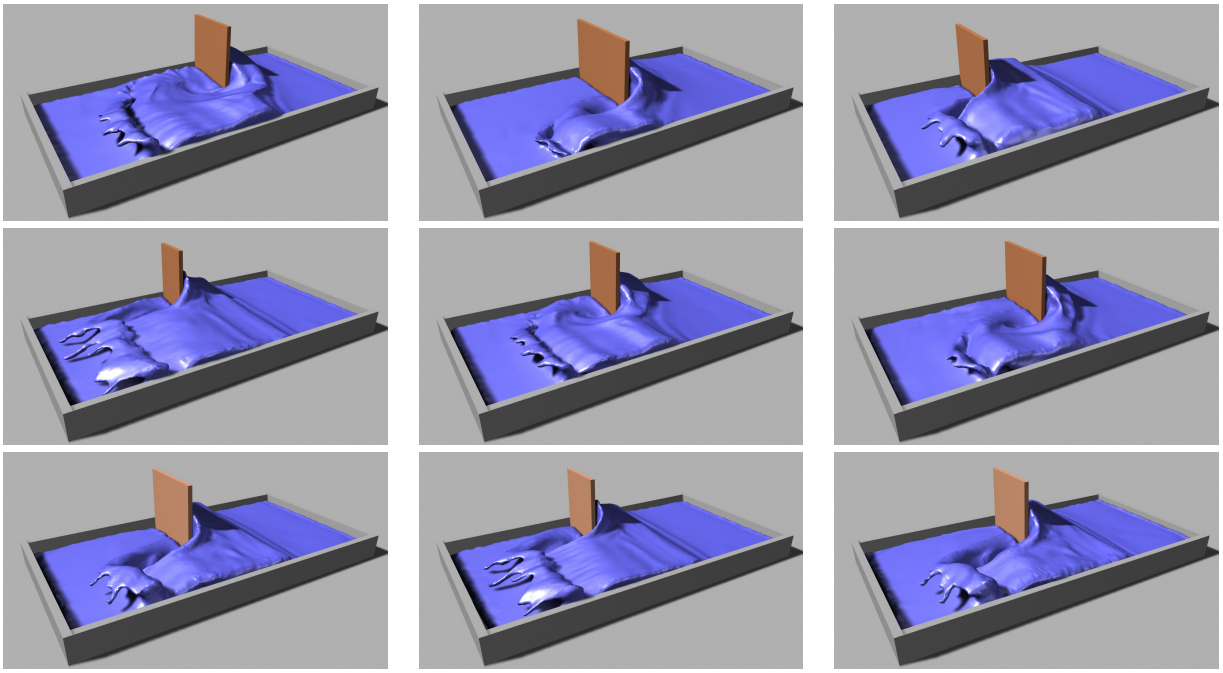

Figure 8: *Snapshots from nine different animations that were created by varying the wall width and wall height of a dam break. Each image is from the same point in time. All results were created by our fluid blending method, and only four original animations were used as input.*

BROCHU, T., BATTY, C., AND BRIDSON, R. 2010. Matching fluid simulation elements to surface geometry and topology. *ACM Trans. Graph. 29*, 4 (July), 47:1–47:9.

BROWN, B., AND RUSINKIEWICZ, S. 2007. Global non-rigid alignment of 3-D scans. *ACM Trans. Graph. 26*, 3 (Aug.).

COHEN-OR, D., SOLOMOVIC, A., AND LEVIN, D. 1998. Three-dimensional distance field metamorphosis. *ACM Trans. Graph. 17*, 2 (Apr.), 116–141.

ENRIGHT, D., NGUYEN, D., GIBOU, F., AND FEDKIW, R. 2003. Using the Particle Level Set Method and a Second Order Accurate Pressure Boundary Condition for Free-Surface Flows. *Proc. Joint Fluids Engineering Conference*.

FOSTER, N., AND METAXAS, D. 1996. Realistic animation of liquids. *Graph. Models Image Process. 58*, 5 (Sept.), 471–483.

GELFAND, N., IKEMOTO, L., RUSINKIEWICZ, S., AND LEVOY, M. 2003. Geometrically stable sampling for the ICP algorithm. In *Int. Conference on 3D Digital Imaging and Modeling (3DIM)*.

GELFAND, N., MITRA, N. J., GUIBAS, L. J., AND POTTMANN, H. 2005. Robust global registration. In *Symposium on Geometry Processing*, Eurographics Association, SGP '05.

HÄHNEL, D., THRUN, S., AND BURGARD, W. 2003. An extension of the ICP algorithm for modeling nonrigid objects with mobile robots. In *Proc. of the Int. Joint Conference on Artificial Intelligence*, IJCAI.

KLEIN, A. W., SLOAN, P.-P. J., FINKELSTEIN, A., AND COHEN, M. F. 2002. Stylized video cubes. In *ACM SIGGRAPH Symposium on Computer Animation*, 15–22.

KWATRA, V., AND ROSSIGNAC, J. 2002. Space-time surface simplification and edgebreaker compression for 2d cel animations. *Int. Journal of Shape Modeling 8*, 2, 119–137.

LI, H., ADAMS, B., GUIBAS, L. J., AND PAULY, M. 2009. Robust single-view geometry and motion reconstruction. *ACM Trans. Graph. 28*, 5 (Dec.), 175:1–175:10.

LI, H., LUO, L., VLASIC, D., PEERS, P., POPOVIĆ, J., PAULY, M., AND RUSINKIEWICZ, S. 2012. Temporally coherent completion of dynamic shapes. *ACM Trans. Graph. 31*, 1 (Feb.), 2:1–2:11.

MCNAMARA, A., TREUILLE, A., POPOVIĆ, Z., AND STAM, J. 2004. Fluid control using the adjoint method. *ACM Trans. Graph. 23* (August), 449–456.

MISZTAL, M. K., ERLEBEN, K., BARGTEIL, A., FURSUND, J., CHRISTENSEN, B. B., BÆRENTZEN, J. A., AND BRIDSON, R. 2012. Multiphase flow of immiscible fluids on unstructured moving meshes. In *Proc. Symposium on Computer Animation*, Eurographics Association, SCA '12, 97–106.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proc. Symposium on Computer Animation*, Eurographics Association, 154–159.

NIELSEN, M. B., AND BRIDSON, R. 2011. Guide shapes for high resolution naturalistic liquid simulation. ACM, New York, NY, USA, vol. 30, 83:1–83:8.

OSHER, S., AND FEDKIW, R. 2002. *Level set methods and dynamic implicit surfaces*. Springer Verlag.

PAN, Z., HUANG, J., TONG, Y., ZHENG, C., AND BAO, H. 2013. Interactive localized liquid motion editing. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 32*, 6 (Nov.).

PAPAZOV, C., AND BURSCHKA, D. 2011. Deformable 3d shape registration based on local similarity transforms. *Computer Graphics Forum 30*, 5, 1493–1502.

RAVEENDRAN, K., THUEREY, N., WOJTAN, C., AND TURK, G. 2012. Controlling liquids using meshes. In *Proce. Symposium on Computer Animation*, Eurographics Association, SCA '12, 255–264.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Efficient variants of the icp algorithm. In *Proc. 3D Digital Imaging and Modeling*, 145–152.

SCHMID, J., SUMNER, R. W., BOWLES, H., AND GROSS, M. 2010. Programmable motion effects. *ACM Trans. Graph. 29*, 4 (July), 57:1–57:9.

SHI, L., AND YU, Y. 2005. Taming liquids for rapidly changing targets. In *Symposium on Computer animation*, ACM, SCA '05, 229–236.

SOLENTHALER, B., AND PAJAROLA, R. 2009. Predictive-corrective incompressible sph. *ACM Trans. Graph. 28*, 3 (July), 40:1–40:6.

STAM, J. 1999. Stable fluids. In *Proc. SIGGRAPH*, ACM, 121–128.

SUMNER, R. W., SCHMID, J., AND PAULY, M. 2007. Embedded deformation for shape manipulation. In *ACM SIGGRAPH 2007 Papers*, ACM, New York, NY, USA, SIGGRAPH '07.

SZELISKI, R. 1996. Matching 3-d anatomical surfaces with non-rigid deformations using octree-splines. *Int. Journal of Computer Vision 18*, 171–186.

TAUBIN, G. 1995. A signal processing approach to fair surface design. In *Proceedings of SIGGRAPH 95*, Annual Conference Series, 351–358.

TEVS, A., BERNER, A., WAND, M., IHRKE, I., BOKELOH, M., KERBER, J., AND SEIDEL, H.-P. 2012. Animation cartography-intrinsic reconstruction of shape and motion. *ACM Transactions on Graphics (TOG) 31*, 2, 12.

THUEREY, N., KEISER, R., RUEDE, U., AND PAULY, M. 2006. Detail-Preserving Fluid Control. *Symposium on Computer Animation* (Jun), 7–12.

TURK, G., AND O'BRIEN, J. F. 1999. Shape transformation using variational implicit functions. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '99, 335–342.

WOJTAN, C., THÜREY, N., GROSS, M., AND TURK, G. 2010. Physics-inspired topology changes for thin fluid features. *ACM Trans. Graph. 29* (July), 50:1–50:8.

YU, J., AND TURK, G. 2010. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, 217–225.

ZENG, Y., WANG, C., WANG, Y., GU, X., SAMARAS, D., AND PARAGIOS, N. 2010. Dense non-rigid surface registration using high-order graph matching. In *IEEE Conf. Computer Vision and Pattern Recognition, CVPR 2010*, 382–389.

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July), 965–972.

13 Appendix

This section describes the details for creating a tetrahedralization of the space-time mesh. Recall that each triangle from one frame has a matching triangle in the next frame, and that connecting these triangles with edges creates a triangular prism. In order to tetrahedralize these prisms, we require that adjacent prisms use the same diagonal edge to split their shared quadrilateral face. That is, the diagonal assignments must be consistent between adjacent prisms. There are two distinct ways to assign diagonals to a single prism. The Type 1 prism can easily be split into just three tetrahedra (see Figure 9, upper left). Unfortunately, the best dissection of the Type 2 prism (Figure 9, upper right) that we have found leads to eight tetrahedra. We form these eight tetrahedra by introducing a new vertex at the center of the prism, and then creating one tetrahedron per triangular face by connecting it to the new center vertex. Since we want to avoid creating more tetrahedra than necessary, we desire an assignment of diagonals to the prisms that includes as many Type 1 prisms as possible, and few or no Type 2 prisms.

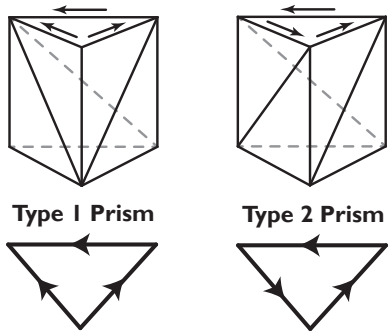

Figure 9: *The two types of triangular prisms, based on the orientation of the diagonals that split their quadrilateral faces. Type 1 prisms (top left) can be divided into three tetrahedra, but Type 2 prisms (top right) require eight. The two triangles (bottom) show the directed edge labels that correspond to the two types of prisms.*

We have devised a simple algorithm that creates a consistent assignment of diagonal edges across all of the prisms, and that also creates only Type 1 prisms. To do this, we only have to consider the triangle mesh from one of the two frames that are to be connected by prisms. We will label each of the edges of this triangle mesh with a directed edge. Each of these directed edges indicates the orientation of a diagonal for the corresponding prism quadrilateral face. Specifically, a directed edge that points towards a particular vertex $\boldsymbol{v}$ indicates that the diagonal edge for that quadrilateral has $\boldsymbol{v}$ as one of its vertices. A triangle that is labeled with edges that are all oriented either clockwise or counter-clockwise (Figure 9, lower right) indicates a Type 2 prism, which is to be avoided. A triangle with mixed orientations indicates a Type 1 prism (Figure 9, lower left), which is desired. Our diagonal assignment algorithm operates purely on the directed edges of the triangle mesh.

To generate the edge directions for a given triangle mesh, we start by assigning a unique numerical label to each vertex of the mesh. Then, each edge of the mesh is assigned the direction that points from the vertex with the lower value towards the vertex with the higher value. This simple rule gives a direction to each edge in the mesh. Note that for any triangle, one of its vertices always has the highest numeric label among the three vertices. This means that two of the directed edges will point towards this vertex, indicating a Type 1 prism (Figure 9, lower left). Since it is not possible for the three vertex indices to form a closed cycle of numeric labels (e.g. $v_1 < v_2 < v_3 < v_1$), a Type 2 configuration (Figure 9, lower right) is impossible. Since we globally assign the numerical vertex labels, adjacent triangles share the same understanding about the direction of their shared edge. This means our diagonal assignment is consistent between adjacent prisms.