6.838: Shape Analysis, 2017
Instructor: Justin Solomon

Fluid Flow Over Mesh Surfaces Via Mesh Parameterization

Malcolm Wetzstein

MIT Department of Electrical Engineering and Computer Science

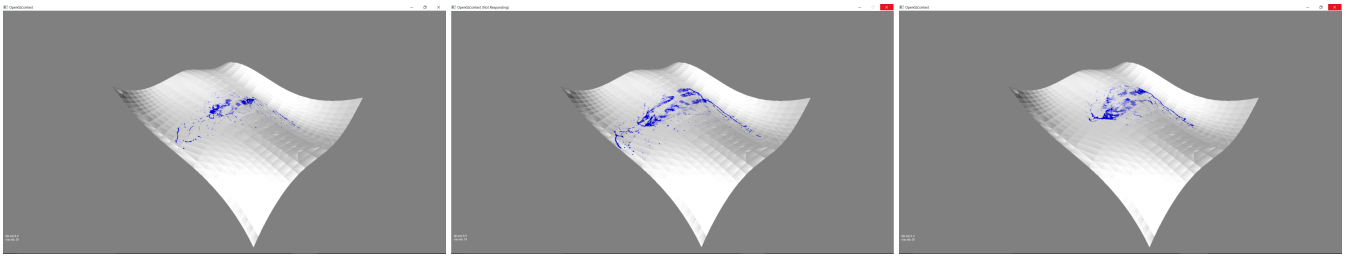

Figure 1: *Comparisons of fluid simulations conducted with the same initial conditions on the same mesh surface using different mesh parameterizations. Left: Fluid simulation using the mesh's original texture coordinates. Center: Fluid simulation using a nearly conformal parameterization for texture coordinates. Right: Fluid simulation using a nearly isometric parameterization.*

Abstract

Quality fluid simulation is a desirable effect in real time applications such as games. Fluid simulation is very computationally expensive however, and in practice real time applications make sacrifices in quality in order to obtain higher performance. This paper investigates a way of increasing the quality of a fluid simulation when the domain of the simulation is restricted to the surface of a mesh. This is done by simulating the fluid flow on a Eulerian grid embedded in the UV texture coordinate space of a mesh, and then using texture coordinates during rendering to query grid cells inside the fluid. Distortion due to the choice of texture parameterization affects the resulting simulation, and we find our tests that an angle preserving parameterization is more effective than an edge length preserving parameterization for fluid simulation on surfaces.

1. Introduction

Real-time applications such as video games often benefit from realistic fluid simulations in order to increase the visual fidelity of liquids in 3D rendered scenes, most commonly water. Accurate fluid simulations are computationally expensive however, prohibitively so for real-time applications. Real-time applications therefore look for appropriate trade-offs between run-time performance and accuracy that best preserve visual fidelity and fit within the application's time constraints. We will explore a special case of fluid simulations in 3D scenes in which less trade-offs can be made between performance and accuracy, allowing a more accurate simulation to still meet real-time performance standards. This special case is fluid flow over surfaces, which are most commonly represented in real-time applications by triangles meshes. By restricting the domain of the simulation to a surface, we can reduce the dimensionality of the fluid simulation problem from simulating in $\mathbb{R}^3$ to simulating in $\mathbb{R}^2$. For Eulerian fluid simulations, an approach to fluid simulation that can more easily track smooth fluid surfaces [BS], we can reduce the run-time of the simulation from order $O(n^3)$ to $O(n^2)$ by reducing the dimensionality of the simulation domain from $\mathbb{R}^3$ to $\mathbb{R}^2$. We introduce a new technique for conducting the dimensionality reduction from three to two dimensions. We accomplish this by parameterizing the mesh representing the surface to the plane by assigning it a set of UV texture coordinates. We then perform a Eulerian fluid simulation in UV texture space, utilizing information about the mesh's normal and tangent data in order to calculate gravitational effects on the fluid. Because planar parameterizations of surfaces in $\mathbb{R}^3$ cause distortion whenever the surface possesses any curvature, the choice of parameterization must be made wisely. Two properties are important for the parameterization to posses in order to prevent distortion, conformality and isometry. A parameterization with no distortion possesses both these properties. When parameterizing a mesh to the plane, choosing to more closely preserve one of these properties will distort the other. It is not clear what balance of preservation of these two properties works best for a given mesh, so we make use of a "hybrid" mesh parameterization technique from [LZY*08] that allows an input parameter λ to adjust along a continuum how conformal versus isometric the re-

sulting mesh parameterization is. λ can be adjusted instead of the mesh parameterization algorithm itself in order to find the parameterization that yields the best visual results for a fluid simulation on a given mesh.

2. Related Work

Similar work has been done in the past in order to simulate fluid flow over surfaces, most of which have chosen triangle meshes as the initial representation of the surface for simulation.

An earlier work utilized Catmull-Clark surfaces to simulate fluid flow over the Catmull-Clark surface representation of a quad mesh [Sta03]. Catmull-Clark surfaces possesses the quality that they are tangent-plane continuous everywhere, and admit a global atlas of parameterizations. This allows fluid flows over surface quad patches to be stitched together without significant distortion. An advantage of this technique over the technique we present is that it allows for surface flow over surfaces of arbitrary topology. Our method uses a mesh parameterization technique that requires the input mesh to have disk topology, a major limitation, though the work of [LPNM02] suggest how one might overcome this limitation (discussed further in section §5). A disadvantage is that the Catmull-Clark surface generated from a mesh may no longer properly represent the surface defined by the mesh, especially if the surface is not intended to be smooth or curved. Our technique works with normal and tangent data of the original mesh, allowing the simulation to remain more accurate to the surface features of the original mesh.

Working directly with triangle meshes, [SY04] discretizes the domain of the mesh surface by defining velocity samples at the centers of each triangle of the mesh and density and pressure samples each each vertex. This method successfully eliminates distortion and guarantees divergence free flows for incompressible fluids. A downside to this technique is that it requires a dense, high-polygon count triangle mesh in order to accurately simulate fluid flow. This is a problem for real time applications that may have tight performance and memory requirements where scenes are created from many different mesh assets and textures that must be cached in memory and/or rendered alongside the simulation. Our technique has the advantage that it supports low-polygon count meshes just as easily due to the simulation being done in UV coordinate space. The quality of our simulation is dictated by the resolution of the Eulerian grid used, independent of the density of the mesh. The technique proposed by [SY04] also had the issue of slow performance at the time of the paper's writing, though modern hardware may alleviate this issue.

Our technique is very similar to that proposed by [HAW*09]. They first parameterize a triangle mesh by computing a conformal mapping of a mesh to the unit cube. Then they simulate an Eulerian fluid simulation over the domain of the surface of the cube. A single parameter related to the conformal mapping is used to modify the PDE equations involved in the fluid simulation in order to account for the distortion caused by the mapping. This technique requires a genus-0 mesh, but mentions how a mesh with disk topology can easily be transformed into a genus-0 mesh. This is a suitable alternative to the technique we present, and a comparison of run-time performance and memory requirements would be a worthwhile investigation to find out which technique may be more suitable for real-time applications. We discuss in section §5 how our technique might be able to use the same method of accounting for distortions in conformal mappings.

A promising technique that uses an alternative approach to simplifying the simulation domain for surfaces is [AMM*12]. The Closest Point Method is used to create an embedding of a triangle mesh into a Eulerian grid in $\mathbb{R}^3$, but only grid cells in the embedding are used, just enough grid cells are used to cover the surface of the mesh. This allows the run-time order to be similar to our technique, they are able to recompute the CPM embedding and forward the fluid simulation each frame in less than 100ms during testing with satisfactory results. The simulation domain does not suffer from distortion like our technique because it is independent of any kind of parameterization. It is a reasonable alternative, though it still may be too slow for some real-time applications.

Techniques proposed by [WLF12] and [AWM*14] simulate directly on triangle meshes, but advance vorticity functions defined on triangle vertices instead of velocity and pressure functions. The performance speed of the technique proposed by [AWM*14] is not suitable for real-time applications. However, the technique of [WLF12] is able to obtain interactive frame rates by using a basis of eigenfunctions to represent the vorticity function. Their technique also has the advantage of working on volume domains built from tetrahedrons, which may have uses for fast general fluid simulations in $\mathbb{R}^3$.

One major advantage of our technique over most of the techniques mentioned above is that it is capable of taking into account artist prescribed surface normals defined at the vertices of the mesh when computing the effects of external forces such as gravity. This allows fluid simulations on low-polygon count models to act on artificial curvature induced upon the mesh by design, much in the same way normals help per-pixel lighting to make flat triangle surfaces appear more curved by interpolating the normals between vertices.

3. Technical Approach

3.1. Overview

There are three main parts to our technique.

The first part is a preprocessing step to create a parameterization of the mesh using the "hybrid" technique from [LZY*08]. The parameterization method takes in a parameter λ that produces a close to conformal or "As-Similar-As-Possible" mapping when $\lambda = 0$, and forces the mapping to become more and more isometric as λ increases, becoming close to isometric or "As-Rigid-As-Possible" as λ goes to infinity.

The second part is another preprocessing step. The Eulerian fluid simulator divides the simulation domain, the UV coordinate space of the parameterized mesh, into a uniform grid, where each grid space is referred to as a grid cell. The fluid solver will require a sample of the mesh's surface normal, tangent, and bitangent (often miscalled binormal in the real-time graphics field) at the center of every grid cell.

The third part is the actual simulation of the fluid in UV coordinate space. This simulation utilizes the UV texture coordinates and the normals, tangents, and bitangents provided by the preprocessing steps in order to conduct a convincing fluid simulation when mapped back to the original mesh surface.

3.1.1. Mesh Parameterization

Suppose we are given a mesh with T triangles. Each triangle $t \in \{1,...,T\}$ is equipped with a local, distortion free, planar parameterization p_t, where p_t has vertices p_t^0, p_t^1, and p_t^2 in the plane. Let u_t be the global, planar parameterization of p_t, subject to distortion. Also let A_t be the area of triangle t in $\mathbb{R}^3$, J_t be the Jacobian matrix that transforms p_t to u_t, and L_t be an "ideal" transformation from p_t to a global parameterization drawn from a family of matrices M. J_t and L_t do not contain rotational components. A common approach to computing a planar parameterization for the mesh is by solving for a parameterization that minimizes the following generic energy equation (taken from [LZY*08]):

$$E(u,L) = \sum_{t=1}^{T} A_t \left|\left|J_t - L_t\right|\right|_F^2 \tag{1}$$

where $||\cdot||_F^2$ refers to the Frobenius norm of a matrix. Note that this method of parameterization requires that every 3D triangle of a mesh first be given a local planar parameterization that preserves all properties of the triangle (angles, area, and edge lengths). We can write this as an optimization problem in the following way:

$$(u,L) = argmin_{(u,L)} E(u,L) \tag{2}$$

$$s.t. L_t \in M$$

where, u gives the desired parameterization of the mesh. It is important to note that two u_t share some vertices when the corresponding triangles share vertices in the input mesh. A matrix L_t is considered ideal in that it receives no penalty for distortion when used to compute the global parameterization of the mesh, ideally $J_t = L_t$ for every triangle. By carefully choosing M, it is possible to choose what property of the mesh's triangles to best preserve in its parameterization.

According to [LZY*08], equation (1) can be rewritten to remove the inconvenient J_t term, replacing it with terms derived from the geometry of the input mesh:

$$E(u,L) = \frac{1}{2}\sum_{t=1}^{T}\sum_{i=0}^{2} \cot(\theta_t^i)\left|\left|(u_t^i - u_t^{i+1}) - L_t(p_t^i - p_t^{i+1})\right|\right|^2 \tag{3}$$

The superscripts are all modulo 2. The p_t^i terms are from the local planar parameterizations of the mesh's 3D triangles discussed earlier, and θ_t^i is the angle opposite edge (p_t^i, p_t^{i+1}) in triangle p_t.

We now consider how a clever choice of M can be used to obtain an approximately conformal parameterization of the mesh onto the plane. In $\mathbb{R}^2$, matrices of the form:

$$M = \left\{ \begin{pmatrix} a & b \\ -b & a \end{pmatrix} : a,b \in \mathbb{R} \right\} \tag{4}$$

represent similarity transforms, which preserve angles. Choosing this as our family of matrices will cause our optimization problem to find an approximately conformal mapping from the input mesh to the plane. Our energy equation is then expressed as:

$$E(u,a,b) = \frac{1}{2}\sum_{t=1}^{T}\sum_{i=0}^{2} \cot(\theta_t^i)\left|\left|(u_t^i - u_t^{i+1}) - \begin{pmatrix} a & b \\ -b & a \end{pmatrix}(p_t^i - p_t^{i+1})\right|\right|^2 \tag{5}$$

This is the energy function for the "As-Similar-As-Possible" parameterization technique mentioned in [LZY*08].

The "hybrid" parameterization technique that we have chose to implement from [LZY*08] uses an energy function that is an extension of the "As-Similar-As-Possible" parameterization technique. Consider matrices of the form:

$$M = \left\{ \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & cos(\theta) \end{pmatrix} : \theta \in \mathbb{R} \right\} \tag{6}$$

which represent rigid transforms that preserve isometry. Referring back to equation (**??**), similarity transforms are a subset of rigid transforms that satisfy:

$$a_t^2 + b_t^2 - 1 = 0 \tag{7}$$

which is only true when $a_t = cos(\theta)$ and $b_t = sin(\theta)$ for some θ. We can now square the left hand side of equation (7) and weight it with a parameter λ, we can use it to devise the following new energy function proposed by [LZY*08]:

$$S(u_t,a_t,b_t) = \sum_{i=0}^{2} \cot(\theta_t^i)\left|\left|(u_t^i - u_t^{i+1}) - \begin{pmatrix} a & b \\ -b & a \end{pmatrix}(p_t^i - p_t^{i+1})\right|\right|^2 \tag{8}$$

$$R(a_t,b_t) = (a_t^2 + b_t^2 - 1)^2 \tag{9}$$

$$E(u,a,b) = \frac{1}{2}\sum_{t=1}^{T}\left[S(u_t,a_t,b_t) + \lambda R(a_t,b_t)\right] \tag{10}$$

The similarity term $S(u_t,a_t,b_t)$ term punishes a triangle in the parameterization for deviating from similarity transforms, as it is taken directly from equation (5) (it is in fact the inner summation). The rigidity term $R(a_t,b_t)$ term is zero when the parameterization of a triangle is rigid, and increases in magnitude as the parameterization becomes less rigid. The rigidity term therefore punishes

triangles for deviating from isometric transforms. The λ parameter can be anywhere in the range $[0, \infty]$ and weights how important it is for the parameterization to be isometric. For $\lambda = 0$, equation (10) reduces to equation (5), and the resulting parameterization is therefore "As-Similar-As-Possible". When $\lambda = \infty$ it is infinitely more important that the parameterization be isometric than conformal, and so the resulting parameterization therefore becomes "As-Rigid-As-Possible". An in between value of λ results in an in between parameterization along the continuous spectrum between "As-Similar-As-Possible" and "As-Rigid-As-Possible".

The work of [LZY*08] shows in their results that the parameterization is very sensitive to small increase in λ, already becoming very rigid for values of λ as small as 0.1. The proper value of λ for a particular mesh may depend on the mesh and need to be experimented for. We share the value of λ that worked best for our testing mesh later in section §4 as a starting place. The parameterization of the mesh is used to assign new UV texture coordinates to the mesh.

3.1.2. Calculating Tangent Space Vectors

The specific tangent and bitangent required by the grid cells of the fluid simulation are those that line up with the U coordinate axis and V coordinate axis respectively in UV coordinate space. The mesh is expected to already be equipped with appropriate normal data. One can average face normals around each vertex if this is not the case. The appropriate tangents and bitangents should be calculated using the mesh's equipped normals and it's UV coordinates (from the parameterization) as described in [Len01]. We explain the reason for this choice of normals, tangents, and bitangents in the next section.

3.1.3. Fluid Simulation

There are several approaches to fluid simulation, most of which either fall into the category of Eulerian or Lagrangian simulations. Some fluid simulation techniques, such as that proposed by [WLF12], do not fall into these categories. To get a quick understanding of Eulerian and Lagrangian fluid simulations and their differences, we refer the reader to [BS]. In short, Eulerian methods track fluid properties on regular grid points throughout the simulation domain, while Lagrangian methods track fluid properties on a collection of particles representing the fluid. To gain a deeper understanding of Eulerian fluid simulations, we refer the reader to [Slo15] and [CM11]. A deeper understanding of Lagrangian fluid simulations should not be necessary for the following sections.

Traditionally real-time graphics applications use Lagrangian methods, specifically Smoothed Particle Hydrodynamics, details of which can be found in [WZQL12]. Lagrangian methods are used for their relatively fast computation time. We choose to use an Eulerian method of fluid simulation however because the computation time of Eulerian simulations improve significantly with the reduction of the dimensionality of the simulation domain from $\mathbb{R}^3$ to $\mathbb{R}^2$, as this reduces the dimensionality of the regular grid involved. Lagrangian methods do not receive as significant a performance boost from this dimensionality reduction. Eulerian simulations have some important benefits over Lagrangian simulations, including better handling of smooth fluid surfaces and higher numerical accuracy [BS]. There are many different fluid simulation techniques that fall under Eulerian methods, but the particular choice of Eulerian simulation technique is not crucial to the surface fluid flow technique we are presenting. Any Eulerian fluid simulation in $\mathbb{R}^2$ can work. We discuss our particular Eulerian technique in section §3.2.3. Two important augmentations are required by the Eulerian fluid simulation however.

The first augmentation required is that each grid cell be equipped with a surface tangent space basis using the normals, tangents, and bitangents from the earlier preprocessing step. Given that basis, the tangent space of the surface is simply the UV coordinate space with an extra vertical dimension, W, dictated by the normal. A tangent space basis built from vectors in the model space of the mesh is required because in real-time applications the external force of gravity that will be acting on the fluid is generally defined in one coordinate space relative to the larger 3D scene being rendered (world space). With each cell equipped with tangent data, we can transform the world space gravitational force to the model space of the surface mesh, and then use the tangent space basis to calculate the magnitude of the U, V, and W components of gravity in tangent space by projecting the gravitational force onto the basis vectors. This gives us the gravitational force for each grid cell in relation to its neighboring cells.

The second augmentation is that there must be a way to convert the fluid's surface representation (the areas of the simulation domain which are considered to be inside the liquid) to an asset that can be rendered over the mesh surface. This will depend on the implementation-specific method of fluid surface tracking. The UV coordinates of the mesh can be used to properly map the liquid surface representation onto the mesh. In section §3.2.4 we give a simple example of how we export our fluid surface tracking description to a texture for the mesh that displays the liquid on it's surface using the UV coordinates from the mesh's parameterization.

3.2. Implementation

This section discusses some important implementation details of the preprocessing steps outlined above. Then we discuss some details of the basic Eulerian fluid solver used in testing our mesh surface fluid flow technique. This is to demonstrate how one can integrate a normal Eulerian fluid simulation in $\mathbb{R}^2$ to simulate flow over a mesh. We implemented the entire algorithm for our mesh surface fluid flow technique in python, using the OpenGL API exposed through PyOpenGL to do 3D rendering of our test scene. Our implementation relies heavily upon the Numpy and Scipy libraries.

3.2.1. Parameterization Details

We have mentioned earlier in section §2 that the "hybrid" approach used in our only works on meshes with disk topology, although we explore later in section §5 ideas for how to overcome this. Alongside the task of processing the mesh to ensure it has disk topology, one should make sure that triangles intended to be connected on the surface indeed share vertices. It is common in real-time graphics applications to allow redundant vertices in order to specify a discontinuous normals, UV coordinates, etc. between adjacent triangles sharing an edge, even if the triangles are intended to be connected. A preprocessing step is required to make sure no redundant

vertices exist in the mesh and that it is fully connected before it is parameterized. The redundant vertices can then be added later and given the UV coordinate of the vertex that was involved in the parameterization so that the vertices match in the fluid simulation.

In order to compute the desired mesh parameterization one must solve the following optimization problem:

$$(u,a,b) = argmin_{(u,a,b)} E(u,a,b) \tag{11}$$

where $E(u,a,b)$ refers to equation (10). The optimization problem of (11) cannot be solved using a system of linear equations, [LZY*08] suggests using an iterative solution that alternates between two steps, reducing the energy of the objective function each time.

The first step is a local phase that minimizes a_t and b_t while holding u_t fixed for each triangle. The following equations are used by [LZY*08] to solve for a_t and b_t for a given triangle t. We omit t subscripts for clarity:

$$\frac{2\lambda(D^2+E^2)}{D^2}a^3 + (C-2\lambda)a - D = 0 \tag{12}$$

$$b = \frac{E}{D}a \tag{13}$$

$$C = \sum_{i=0}^{2} \cot(\theta_t^i)\left[(p_x^i - p_x^{i+1})^2 + (p_y^i - p_y^{i+1})^2\right]$$

$$D = \sum_{i=0}^{2} \cot(\theta_t^i)\left[(u_x^i - u_x^{i+1})(p_x^i - p_x^{i+1}) + (u_y^i - u_y^{i+1})(p_y^i - p_y^{i+1})\right]$$

$$E = \sum_{i=0}^{2} \cot(\theta_t^i)\left[(u_x^i - u_x^{i+1})(p_y^i - p_y^{i+1}) - (u_y^i - u_y^{i+1})(p_x^i - p_x^{i+1})\right]$$

The p and u are the initial local and current global parameterizations of a triangle respectively, similar to the p_t and u_t terms in equation (3). Equation (14) is a cubic equation with three roots, bringing into question which root should be used as the solution to the system of equations. No advice from [LZY*08] is given on which root to choose, however in practice we found that equation (14) tends to only have one real solution until λ gets close to 1.0 (multiple real solutions appeared somewhere around a value of $\lambda = 0.8$ for our test mesh). This leaves more than enough room for adjusting λ to pick a mostly rigid parameterization before one has to start deciding which root to take. We do experiment with λ values over 0.8, and in such cases we simply pick an arbitrary real solution (the first one returned by Numpy's polynomial root solver).

The second step is a global phase. We hold a and b constant and solve globally for the u that minimizes the energy function. Notice that when a and b are held constant, the derivatives of the rigidity term of equation (10) with respect to the components of u are all zero. Therefore we can ignore the rigidity component in the global step. Taking the derivative of the similarity component of equation (10) gives a system of linear equations, one equation for each component of u. Solving this system for a new set of u components lowers the energy of the objective function.

An expression for the derivatives of an energy function of the same form as equation (8) is given by [LZY*08]:

$$\sum_{j\in N(i)} \left[\cot(\theta_{ij}) + \cot(\theta_{ji})\right](u_i - u_j) =$$
$$\sum_{j\in N(i)} \left[\cot(\theta_{ij}) \begin{pmatrix} a_{t(i,j)} & b_{t(i,j)} \\ -b_{t(i,j)} & a_{t(i,j)} \end{pmatrix} + \cot(\theta_{ji}) \begin{pmatrix} a_{t(i,j)} & b_{t(i,j)} \\ -b_{t(i,j)} & a_{t(i,j)} \end{pmatrix}\right](p_i - p_j) \tag{14}$$

where each i is a vertex of a triangle, $N(i)$ are its neighboring vertices, θ_{ji} is the angle opposite the directed half-edge (i,j), and $a_{t(i,j)}$ and $b_{t(i,j)}$ are the a_t b_t of the triangle containing the half-edge (i,j). The coefficients of this system of linear equations only depend on the geometry of the input mesh, and so the same matrix can be reused every global phase. Taking advantage of matrix pre-factoring can significantly improve the efficiency of this pre-processing step.

This iterative method of mesh parameterization requires an initial parameterization of the mesh. Other mesh parameterizations that did not require initial parameterizations were used by [LZY*08] to seed the "hybrid" technique. We elected to use texture coordinates provided by the software used to create our testing meshes, Rhino, in order to seed our implementation.

The resulting parameterization of the "hybrid" technique may extend outside of the $[0.0, 1.0]$ range of UV coordinates. In this case, the parameterization should be uniformly scaled to fit snuggly within the unit square so that it can be used as a set of UV texture coordinates for the mesh.

3.2.2. Computing Tangents for Each Grid Cell

Following [Len01], we can compute the tangent $\boldsymbol{T}$ and bitangent $\boldsymbol{B}$ for a given triangle by solving the following system of equations:

$$\boldsymbol{Q_1} = s_1\boldsymbol{T} + t_1\boldsymbol{B}$$

$$\boldsymbol{Q_2} = s_2\boldsymbol{T} + t_2\boldsymbol{B}$$

$$\boldsymbol{Q_1} = \boldsymbol{P_1} - \boldsymbol{P_0}$$

$$\boldsymbol{Q_2} = \boldsymbol{P_2} - \boldsymbol{P_0}$$

$$(s_1,t_1) = (u_1 - u_0, v_1 - v_0)$$

$$(s_2,t_2) = (u_2 - u_0, v_2 - v_0)$$

where $\boldsymbol{P_0}, \boldsymbol{P_1}, \boldsymbol{P_2}$ are the vertices of the triangle and $(u_0,v_0),(u_1,v_1),(u_2,v_2)$ are the UV texture coordinates of each vertex. An explicit solution is given in code by [Len01] for this

system of equations. A tangent or bitangent for a vertex is computed similarly to a normal, the tangent or bitangent for each adjacent triangle is accumulated at the vertex and then averaged by normalizing the end sum. There is one issue with this approach. We mentioned earlier in section §3.2.1 that redundant vertices are commonly used in real-time applications, especially when discontinuities in normals or tangents are desired between triangles. Before calculating tangents and bitangents, redundant vertices should be reintroduced between triangles that are meant to have tangent discontinuities. This will prevent tangents from averaging onto the same vertex and inducing continuous tangents at triangle edges intended to be creases. Redundant vertices should all share the same UV coordinate provided by the previous mesh parameterization step.

Once they are all computed, normals, tangents, and bitangents are only defined at the vertices of each of the mesh's triangles (including triangles formed by redundant vertices). We now need to provide interpolated values to each grid cell of the fluid simulator. The UV are the embedding of the triangles in the plane by the earlier mesh parameterization, this planar UV coordinate space is also where the grid cells reside. We assign normals, tangents, and bitangents to the grid cells by treating the grid cells as pixels in a buffer and rasterizing the triangles in UV coordinate space using their UV coordinates as the vertices. The normals, tangents, and bitangents are interpolated to grid cells inside the rasterized triangle. We define the screen space of this buffer containing the grid cells to be the same as UV coordinate space, allowing us to use a very rudimentary software rasterizer to rasterize the UV triangles (no projections or transformations involved, only conversions between grid indices and UV coordinates; the software rasterizer is less than 50 lines of python code).

3.2.3. Our Fluid Solver

Generally when Eulerian fluid simulation methods are used in real time, a solver for the fluid simulation is implemented on the GPU in order to take advantage of the high parallelism of Eulerian algorithms. This is usually a requirement in order to achieve frame rates suitable for real-time graphics applications, although an efficient CPU implementation may be able to achieve interactive frame rates. Our technique would require a GPU implementation of a Eulerian fluid solver in order to be effectively utilized in it's target use case, real-time applications such as games. However, due to complete lack of past experience in implementing fluid solvers, and the difficulty of implementing them, we have chosen to do a CPU implementation of our fluid solver in Python.

The fluid solver we have implemented uses very basic techniques. We discretize the plane and track velocity and pressure samples according to the MAC cell discretization described in [BS]. Each grid cell keeps track of a scalar pressure sample at it's center. Samples of velocities in the U-axis direction are tracked as scalars on the midpoints of horizontal (left and right) faces between cells, and samples of velocities in the V-axis direction are tracked as scalars on the midpoints of vertical (top and bottom) faces between cells. The U and V axes mentioned are the axes of the UV coordinate space.

A crucial part of Eulerian fluid simulations is the method used to track the surface of the fluid, in other words, which grid cells are considered inside the fluid. Our fluid solver uses marker particles, massless, volume-less particles scattered throughout the simulation domain that flow along the velocity field of the fluid. Cells containing marker particles are considered inside the fluid, all other cells are outside. The use of marker cells provides a simple method for injecting fluid into the simulation, we simply create marker particles around the source of the fluid. Marker particles disappear when they flow outside the domain of the fluid simulation.

For readers familiar with Eulerian fluid simulations, we give a brief overview of how each frame is computed in our fluid solver (please refer to [BS], [CM11], and [Slo15] for a more thorough explanation of how these steps are implemented). First fluid velocity is advected along flow line using an Runge-Kutta 3rd order time integrator to step backwards along the velocity field to sample fluid velocity upstream and advect it forward. Then external forces are applied to the advected velocity field. The only external force we consider is gravity. For each velocity sample, we interpolate the normal, tangent, and bitangent data from the centers of the cells on either side of the velocity sample. We then use the normal at the velocity sample to subtract out the component of the gravitational vector perpendicular to the surface, we assume this component has no effect on the fluid. We are left with the tangential component of gravity. If working with a horizontal velocity sample in the U-axis direction, we project the tangential gravity onto the tangent vector at the sample location (the tangent being interpolated from neighboring cells). If working with a V axis velocity we do the same but with the bitangent. The final axis-aligned component of the gravitational vector is used to update the velocity sample. After external forces are handled, we solve the Poisson equation:

$$\nabla^2 p = \frac{\rho}{\delta t} \nabla \cdot v \tag{15}$$

where p is the pressure inside a cell, v is the velocity sample inside a cell, ρ is the density of the fluid, and δt is the length of the current time step. MAC cells provide a convenient way of solving the Poisson equation using a system of linear equations derived from a discretization of the laplacian on the MAC cells. This is done in order to solve for the correct pressure values in every cell that will force the velocity field to become divergence free, a property of incompressible liquids such as water. Lastly, we advect marker particles along the velocity field of the simulator using a simple forward Euler time integrator.

3.2.4. Rendering the Fluid

This section describes a simple method of rendering the fluid on the 3D mesh. To do this, we create an image texture with a resolution matching the dimensions of the Eulerian grid. We then query each grid cell to see if it contains a marker particle, i.e. it is inside the liquid. If this is true, we set the color of the corresponding pixel in the image texture to the color of the fluid. Pixels corresponding to cells without marker particles are set to white. Because the Eulerian grid is in UV space, we have just created a texture that already maps an image of the fluid directly onto the 3D mesh, and we use it during rendering to modulate the material color of the mesh. Alternatively, cells outside the liquid can correspond to clear cells and

the fluid image can be alpha blended with an underlying texture. Our implementation extends this method to color pixels in the fluid image based on the number of marker particles in a grid cell, possibly setting the color of the pixel to an intermediate color in areas where less marker particles tend to travel.

4. Results

Our experience using the "hybrid" parameterization technique was somewhat different than in [LZY*08]. By using approximately conformal parameterizations to seed the "hybrid" technique, they were able to achieve fast convergence, within 10 iterations or less for most meshes. The mesh used for testing in our experiments was created in Rhino, a 3D modeling software program. We do not know what kind of mesh parameterization technique they use to produce texture coordinate. For our mesh, many more than 10 iterations were needed to achieve convergence, and we ended up using 100 iterations to guarantee convergence. The time needed to perform 100 iterations on our mesh is roughly 12-13 seconds. Our testing mesh has

Our CPU implementation of a fluid solver is able to simulate a little over 1 frame per second for a 256 by 256 Eulerian grid resolution, barely fast enough to interact with and unsuitable for any real-time application. If we move the same algorithm to the GPU, it would not be unreasonable to expect at least a 100x speedup, guaranteeing us real-time frame-rates with a relatively naive implementation.

We experimented with different values of λ to obtain different results with our fluid simulation. The values we experimented with were $\{0.0, 0.001, 0.1, 1.0, 10000.0\}$. We also experimented with the original parameterization of the mesh provided by Rhino. Figures $2-7$ are visualizations of the parameterizations of the mesh for the different choices of λ. Figures $8-13$ show the results of fluid simulations on the different mesh parameterizations.

From observing fluid flows, it is clearly evident that very conformal maps suffer from unnatural fluid speedups due to distortion in the size of the parameterized triangles. The triangle shrinks in relation to the other triangles in UV space and the fluid is able to more quickly pass over that triangle due to its shrinking. This is evident in the pictures by the fact that particles in simulations utilizing more conformal mappings have a much easier time reaching the edge of the simulation domain from the center. This is especially easy to notice when watching the simulation. The original Rhino parameterization seems to suffer some of the same artifacts as the conformal parameterizations. The very rigid mappings pose a different issue that is harder to see, but is still evident in the pictures. The fluid in more rigid mappings sometimes seems to get stuck on slopes and does not always collect and pool up where the surface is most level. I do not have a good intuition for the reason for this distortion. Our tests suggest that a parameterization that takes on a bit of rigidness works best, just enough to reign in the size distortion of the parameterization. My recommendation would be to start with a value of 0.01.

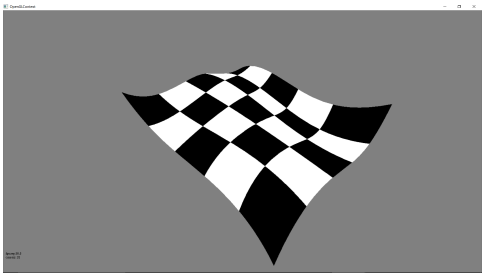

Figure 2: *This is a texture mapped onto the testing mesh with its original texture map.*

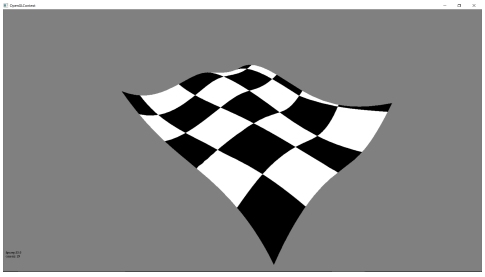

Figure 3: *Parameterization with $\lambda = 0$, an "As-Similar-As-Possible" parameterization.*

5. Future Work

There are many possible extensions of the technique presented in this paper. Two future extensions are important in particular. To handle arbitrary meshes with our method, they must be cut or segmented into components possessing disk topology. [LZY*08] mentions how some meshes can be given disk topology by cutting along a seam. Segmentation of meshes with conformal mappings is discussed in [LPNM02], along with methods for efficiently packing them in one texture. What would need to be added to these techniques is a way of connecting grid cells in UV coordinate space that lie on either side of a cutting seam, most likely through pointers. It is not obvious whether or not such seam line cells would have orientations with faces that are parallel when matched across the seam, and this would have to be investigated. Another important extension is to use methods to compensate for the effects of distortions of the mesh surface in UV coordinate space. There is an effective solution for this problem presented in [HAW*09] that could reasonably be incorporated into our method, but is specific to conformal mappings only. We would have to experiment if it is still an effective correction for more rigid mappings at all. Another easily done, cool extension would be to support normal (bump) mapping in the fluid simulation, since a tangent space basis is already defined at each grid cell. This could allow surfaces defined by low polygon count meshes to be arbitrarily bumpy. This technique requires more work and testing in general to full evaluate its potential. Tests on more meshes should be conducted, and the fluid solver should be implemented on the GPU. I also think that the technique already has the ability to work on meshes undergoing transformations in

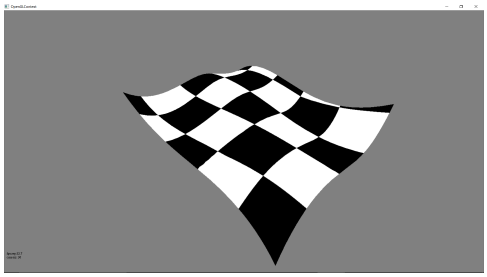

Figure 4: *Parameterization with* $\lambda = 0.001$.

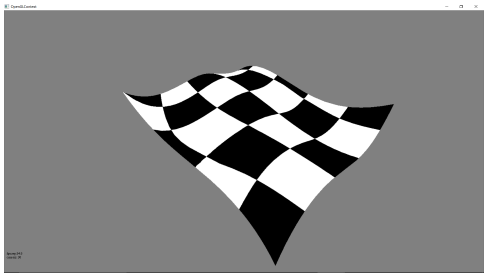

Figure 5: *Parameterization with* $\lambda = 0.1$.

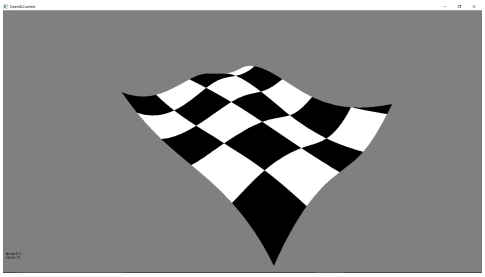

Figure 6: *Parameterization with* $\lambda = 1.0$.

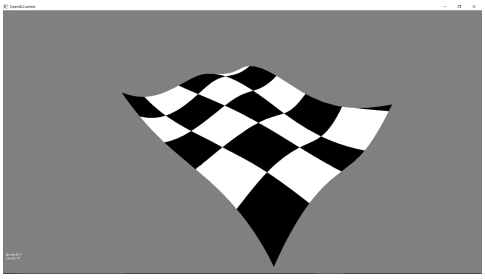

Figure 7: *Parameterization with* $\lambda = 10000.0$*, close to an "As-Rigid-As-Possible" parameterization.*

real time, reacting to rotations of the object relative to gravity, but this still needs to be tested.

6. Conclusion

We have developed an interesting new technique for simulating fluid flow over mesh surfaces. Fluid flow over mesh surfaces is not a new endeavor, but this technique is unique in that it aims to be highly useful and easily integrated into real-time graphics applications, such as video games. There is still much more work to be done before the technique's potential can be realized.

References

[AMM*12] AUER S., MACDONALD C. B., MARC T., SCHNEIDER J., WESTERMANN R.: Real-time fluid effects on surfaces using the closest point method. *Computer Graphics Forum 31*, 6 (Sept. 2012), 1909–1923. 2

[AWM*14] AZENCOT O., WEIÃ§MANN S., MAKS O., WARDETZKY M., BEN-CHEN M.: Functional fluids on surfaces. *Computer Graphics Forum 33*, 5 (Aug. 2014), 237–246. 2

[BS] BRALEY C., SANDU A.: Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. 1, 4, 6

[CM11] CHENTANEZ N., MÃIJLLER M.: Real-time eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics 30*, 4 (July 2011), 82:1âĂŞ–82:10. 4, 6

[HAW*09] HEGEMAN K., ASHIKHMIN M., WANG H., QIN H., GU X.: Gpu-based conformal flow on surfaces. *Communications in Information and Systems 9*, 2 (Dec. 2009), 197–212. 2, 7

[Len01] LENGYEL E.: Computing tangent space basis vectors for an arbitrary mesh, 2001. URL: http://www.terathon.com/code/tangent.html. 4, 5

[LPNM02] LÃL'VY B., PETITJEAN S., NICOLAS R., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics 21*, 3 (July 2002), 362–371. 2, 7

[LZY*08] LIU L., ZHANG L., YIN X., GOTSMAN C., GORTLER S. J.: A local/global approach to mesh parameterization. *Computer Graphics Forum 27*, 5 (July 2008), 1495–1504. 1, 2, 3, 4, 5, 7

[Slo15] SLOCUM J.: Efficient fluid simulation algorithms, Apr. 2015. 4, 6

[Sta03] STAM J.: Flows on surfaces of arbitrary topology. *ACM Transactions on Graphics 22*, 3 (July 2003), 724–731. 2

[SY04] SHI L., YU Y.: Inviscid and incompressible fluid simulation on triangle meshes. *Computer Animation and Virtual Worlds 15*, 3-4 (June 2004), 173–181. 2

[WLF12] WITT T. D., LESSIG C., FIUME E.: Fluid simulation using laplacian eigenfunctions. *ACM Transactions on Graphics 31*, 1 (Jan. 2012), 1–11. 2, 4

[WZQL12] WANG W., ZHONGZHOU J., QIU H., LI W.: Real-time simulation of fluid scenes by smoothed particle hydrodynamics and marching cubes. *Mathematical Problems in Engineering 2012* (Oct. 2012), 1–9. 4

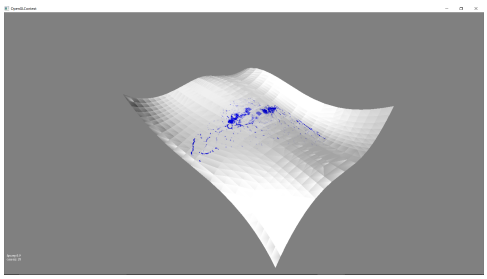

Figure 8: *Fluid flow performed on the testing mesh with its original texture map.*

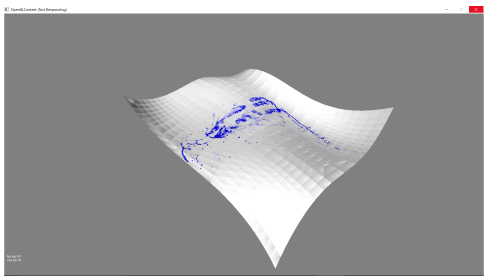

Figure 9: *Fluid flow on a parameterization with* $\lambda = 0$*, an "As-Similar-As-Possible" parameterization.*

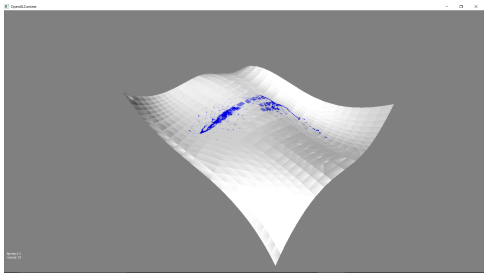

Figure 10: *Fluid flow on a parameterization with* $\lambda = 0.001$.

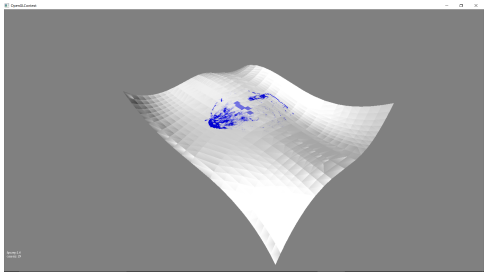

Figure 11: *Fluid flow on a parameterization with* $\lambda = 0.1$.

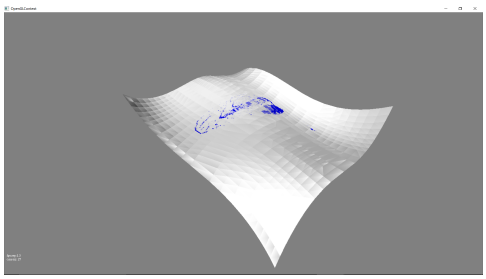

Figure 12: *Fluid flow on a parameterization with* $\lambda = 1.0$.

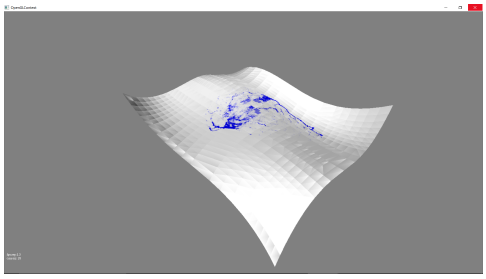

Figure 13: *Fluid flow on a parameterization with* $\lambda = 10000.0$*, close to an "As-Rigid-As-Possible" parameterization.*