Milestone Report: Rendering Dye Advection in Three-Dimensional Fluids

Ethan Li
CS 148 (Summer 2016) Final Project
August 2, 2016

Table of contents

1 Summary

This report discusses my achievement of Milestone 1 and my ongoing progress in Milestone 2.

Milestone 1 focuses on achieving a basic implementation of the simulation software. Milestone 2 focuses on upgrading the complexity of the fluid model for better visuals. Milestone 3 focuses on improving the accuracy of the simulation algorithm. Milestone 4 (previously proposed as part of Milestone 2) focuses on providing user interactions with the fluid model. Milestone 5 focuses on visual refinement producing some animation renders using the software. Optional Milestone 6 focuses on improving fluid rendering for better visuals. Optional Milestone 7 focuses on porting the fluid simulation to fragment shaders, potentially providing a performance boost over the CPU or at least tolerable performance in WebGL.

2 Milestone 1

This milestone consists of:

1. Understanding the mathematics underlying fluid dynamics simulation sufficiently well to implement fluid simulation algorithms.

2. Reimplementing Jos Stam's Stable Fluids algorithm to work in C++ with the Eigen linear algebra library and the programmable OpenGL pipeline

2.1 Mathematics

The summary of mathematics in this subsection is a review of Jos Stam's work on his Stable Fluids algorithm ([4] and [5]), Dan Morris's notes on Stam's original work [1], and Mark Harris's discussion of Stam's work [2].

2.1.1 Summary of the Navier-Stokes Equations

The Navier-Stokes equations model fluid flow. The equations describing the evolution over time of velocity $\vec{u}$ and pressure p can be formulated as follows:

$$\nabla \cdot \vec{u} = 0 \quad (1)$$

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\,\vec{u} - \frac{1}{\rho}\nabla p + \nu\,\nabla^2\,\vec{u} + \vec{f} \quad (2)$$

where ν is kinematic viscosity, ρ is fluid density, and $\vec{f}$ is an external force. In three dimensions, recall that $\nabla p = \begin{pmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial x} \end{pmatrix}$ (gradient), $\nabla \cdot \vec{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$ (divergence), and $\nabla^2\,\vec{u} = \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2}$ (Laplacian).

Equation 1 corresponds to conservation of mass for an incompressible fluid. Equation 2 consists of terms capturing the advection of velocity, or the transport of velocity along the velocity field; generation of velocity by pressure differences, which is zero for incompressible fluids; diffusion of velocity, or the dissipation of movement; and forces applied to the fluid. Note that the first term is non-linear in $\vec{u}$, which makes the Navier-Stokes equations difficult to solve. With Neumann boundary conditions, the velocity field at system boundaries ∂D should have a perpendicular component of zero.

The scalar field ρ of the density of dye in the fluid (or indeed of any other quantity of the fluid) can then be modeled as follows:

$$\frac{\partial \rho}{\partial t} = -(\vec{u} \cdot \nabla)\,\rho + \kappa\,\nabla^2\,\rho + S \quad (3)$$

where κ is the diffusion constant of the dye. Similarly to the terms in equation 2, these terms capture advection of dye, diffusion of dye, and an external source of dye. Note that the transport of distinct substances can modeled with additional scalar fields, diffusion constants, and equations.

Then fluid dynamics simulation is a matter of developing and using some numerical method to compute the transport of fluid velocity and dye density according to these equations at successive time steps.

2.1.2 Computing Transport of Fluid Velocity

Given initial conditions $\vec{u}^{(0)} = \vec{u}(\vec{x}, 0)$, we wish to compute $\vec{u}(\vec{x}, t + \Delta t)$ from $\vec{u}(\vec{x}, t)$.

Stam presents an unconditionally stable implicit method for accomplishing this task. A key element of this method is the application of the Helmholtz-Hodge Decomposition to enforce equation 1. Any vector field $\vec{w}$ can be decomposed (by Helmholtz-Hodge) into

$$\vec{w} = \vec{u} + \nabla p \quad (4)$$

where $\nabla \cdot \vec{u} = 0$ and p is a scalar field. Then $\vec{u}$ can be computed as the projection $\mathbf{P}$ of $\vec{w}$, such that $\vec{u} = \mathbf{P}\,\vec{w}$. $\mathbf{P}$ is defined from applying ∇ to equation 4 as $\nabla \cdot \vec{w} = \nabla \cdot \vec{u} + \nabla^2 p = \nabla^2 p$ (the Poisson equation for p with Neumann boundary conditions); then solving the Poisson equation $\nabla \cdot \vec{w} = \nabla^2 p$ for p given $\vec{w}$ gives $\mathbf{P}\,\vec{w} = \vec{w} - \nabla p$. Equation 1 gives $\mathbf{P}\,\vec{u} = \vec{u}$; and $\nabla p = \vec{0} + \nabla p$ gives $\mathbf{P}\,\nabla p = \vec{0}$. Then applying projection to equation 2 results in a single equation for the evolution of velocity, the Poisson-pressure equation:

$$\frac{\partial \vec{u}}{\partial t} = \mathbf{P}\left(-(\vec{u} \cdot \nabla)\,\vec{u} + \nu\,\nabla^2\,\vec{u} + \vec{f}\right)$$

So temporary introduction of nonzero $\nabla \cdot \vec{u}$ (in violation of equation 1) by simulation computations, namely adding $\vec{f}$, advecting, and diffusing (in that order) resulting in intermediate computation result $\vec{w}$, can be corrected by applying a projection to $\vec{w}$ afterwards. Then we compute $\vec{u}(\vec{x}, t + \Delta t)$ from $\vec{u}(\vec{x}, t)$ as follows:

$$\begin{aligned}
\vec{w}^{(0)}(\vec{x}, t + \Delta t) &\leftarrow \vec{u}(\vec{x}, t) \\
\vec{w}^{(1)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Add Forces To}\,\vec{w}^{(0)}(\vec{x}, t + \Delta t) \\
\vec{w}^{(2)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Advect}\,\vec{w}^{(1)}(\vec{x}, t + \Delta t)\,\textbf{along}\,\vec{w}^{(1)}(\vec{x}, t + \Delta t) \\
\vec{w}^{(3)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Diffuse}\,\vec{w}^{(2)}(\vec{x}, t + \Delta t) \\
\vec{w}^{(4)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Project}\,\vec{w}^{(3)}(\vec{x}, t + \Delta t) \\
\vec{u}(\vec{x}, t + \Delta t) &\leftarrow \vec{w}^{(4)}(\vec{x}, t + \Delta t)
\end{aligned}$$

where $\vec{w}^{(i)}(\vec{x}, t + \Delta t)$ are temporary intermediate computation results in the current simulation step.

Adding Forces

We assume constant forces with respect to time in any given timestep at the time scale of Δt. If the forces for the current timestep Δt are given by $\vec{f}(\vec{x}, t + \Delta t)$, then

$$\vec{w}^{(1)}(\vec{x}, t + \Delta t) \leftarrow \vec{w}^{(0)}(\vec{x}, t + \Delta t) + \Delta t\,\vec{f}(\vec{x}, t + \Delta t)$$

Advection

Stam uses the method of characteristics from PDE theory in order to stably solve the advection step. Intuitively, this can be summarized as calculating the new velocity $\vec{w}^{(2)}(\vec{x}, t + \Delta t)$ for a virtual particle at $\vec{x}$ as the velocity $\vec{w}^{(1)}(\vec{s}, t + \Delta t)$ at the location $\vec{s}(\vec{x}, -\Delta t)$ obtained by tracing that virtual particle at $\vec{x}$ back in time by Δt along $\vec{w}^{(1)}(\vec{x}, t + \Delta t)$. Because this approach relies on tracing the flow of virtual particles, it is considered semi-Lagrangian. In general, we have

$$\vec{w}^{(2)}(\vec{x}, t + \Delta t) \leftarrow \vec{w}^{(1)}(\textbf{Trace}\,\vec{x}\,\textbf{Along}\,\vec{w}^{(1)}(x, t + \Delta t)\,\textbf{and}\,{-\Delta t}, t + \Delta t)$$

If we use the first-order approximation $\vec{s}(\vec{x}, -\Delta t) \approx \vec{x} - \vec{w}^{(1)}(x, t + \Delta t)\,\Delta t$, then we have

$$\vec{w}^{(2)}(\vec{x}, t + \Delta t) \leftarrow \vec{w}^{(1)}(\vec{x} - \vec{w}^{(1)}(x, t + \Delta t)\,\Delta t, t + \Delta t)$$

Note also that if $\vec{u}$ is discretized on a rectangular grid, then $\vec{s}(\vec{x}, -\Delta t)$ can be trilinearly interpolated (or bilinearly interpolated in a two-dimensional grid) from velocities at the eight (or four in a two-dimensional grid) nearest grid points.

Diffusion

Stam uses an implicit method to solve the diffusion equation for velocity, $\frac{\partial \vec{w}^{(2)}}{\partial t} = \nu \nabla^2 \vec{w}^{(2)}$:

$$(I - \nu\, \Delta t\, \nabla^2)\vec{w}^{(3)}(\vec{x}, t + \Delta t) = \vec{w}^{(2)}(\vec{x}, t + \Delta t) \tag{5}$$

where I is the identity operator. If ∇^2 is discretized, then I can also be represented as the identity matrix, and we can solve this equation as a sparse linear system for $\vec{w}^{(3)}(\vec{x}, t + \Delta t)$.

Projection

To project $\vec{w}^{(3)}(\vec{x}, t + \Delta t)$, we use a Poisson solver to solve $\nabla^2 p = \nabla \cdot \vec{w}^{(3)}(\vec{x}, t + \Delta t)$ for p, so that

$$\vec{w}^{(4)}(\vec{x}, t + \Delta t) \leftarrow \vec{w}^{(3)}(\vec{x}, t + \Delta t) - \nabla p$$

Note that, if $\vec{u}$ is discretized on a rectangular grid, then $\nabla^2 p = \nabla \cdot \vec{w}^{(3)}(\vec{x}, t + \Delta t)$ is also a sparse linear system.

2.1.3 Computing Transport of Dye

In each simulation step, we calculate the evolution of the fluid's velocity field $\vec{u}(\vec{x}, t + \Delta t)$ from $\vec{u}(\vec{x}, t)$ and then we calculate the evolution of the dye's density field $\rho(\vec{x}, t + \Delta t)$ from $\rho(\vec{x}, t)$ and $\vec{u}(\vec{x}, t + \Delta t)$.

Evolution of the density field ρ of dye follows a very similar structure as with the velocity field of the fluid. We compute $\rho(\vec{x}, t + \Delta t)$ from $\rho(\vec{x}, t)$ as follows:

$$\begin{aligned}
\sigma^{(0)}(\vec{x}, t + \Delta t) &\leftarrow \rho(\vec{x}, t) \\
\sigma^{(1)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Add Density To}\, \sigma^{(0)}(\vec{x}, t + \Delta t) \\
\sigma^{(2)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Advect}\, \sigma^{(1)}(\vec{x}, t + \Delta t)\, \textbf{along}\, \vec{u}(\vec{x}, t + \Delta t) \\
\sigma^{(3)}(\vec{x}, t + \Delta t) &\leftarrow \textbf{Diffuse}\, \sigma^{(2)}(\vec{x}, t + \Delta t) \\
\rho(\vec{x}, t + \Delta t) &\leftarrow \sigma^{(3)}(\vec{x}, t + \Delta t)
\end{aligned}$$

2.1.4 Discretizing the System on a Grid

We represent ρ, S, and the individual components of $\vec{u}$ and $\vec{f}$ as values on a rectangular grid G of cells, where field values are defined at the center of each cell. The grid has N_d cells and a length of L_d along axis $d \in \{x, y, z\}$. Then each cell has a length of $l_d = \frac{L_d}{N_d}$ along axis d. If the grid has origin $\vec{0}$ corresponding to grid cell $G_{0,0,0}$, then the center of grid cell $G_{i,j,k}$ has location $\vec{x} = (\vec{X} + 0.5) \circ l = \begin{pmatrix} (i + 0.5)\, l_x \\ (j + 0.5)\, l_y \\ (k + 0.5)\, l_z \end{pmatrix}$ where $\vec{X} = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$ and i, j, and k are nonnegative integers. Additionally, for a given integer x_d, the indices X_d of the nearest cells along axis d are given by $\frac{x_d}{l_d} - 0.5$; and for a given non-integer x_d, they are given by $\left\lfloor \frac{x_d}{l_d} - 0.5 \right\rfloor$ and $\left\lfloor \frac{x_d}{l_d} - 0.5 \right\rfloor + 1$.

The **Add Forces** and **Add Density** steps are merely cell-wise addition of $\Delta t\, \vec{f}$ to $\vec{u}$ and of S to ρ.

$\textbf{Advect}\, V\, \textbf{along}\, U$ (where V is the scalar or vector field to advect along vector field U) computes $\vec{x}(i, j, k)$ for each (i, j, k) and trilinearly interpolates the value of V at $\textbf{Trace}\, \vec{x}(i, j, k)\, \textbf{Along}\, \vec{w}^{(1)}(x, t + \Delta t)\, \textbf{and} - \Delta t$. We can implement $\textbf{Trace}\, \vec{x}\, \textbf{Along}\, U\, \textbf{and} - \Delta t$ as a first-order approximation (as discussed above and presented in [5]), or as a second-order Runge-Kutta approximation (as claimed in [4]).

The same sparse linear solver can be used for diffusion and projection. It can be implemented as an iterative (relaxation) method such as Gauss-Seidel (as used in [5]), Jacobi, or Conjugate Gradient iteration, or as a multi-grid algorithm (the best theoretical choice but slow in practice, according to [4]).

To prepare the diffusion equation 5 for such a solver, we discretize it on a grid. For a vector or scalar field V, the diffusion step $V(\vec{x}, t+\Delta t) \leftarrow \textbf{Diffuse}\, V(\vec{x}, t)$ obeys the equation

$$V(\vec{x}, t+\Delta t) - \nu\, \Delta t\, \nabla^2 V(\vec{x}, t+\Delta t) = V(\vec{x}, t)$$

Let $V_{i,j,k} \leftarrow V(\vec{x}, t)$ and $V'_{i,j,k} \leftarrow V(\vec{x}, t+\Delta t)$. Then the equation is discretized as

$$V'_{i,j,k} - \nu\, \Delta t \left(\frac{V'_{i+1,j,k} + V'_{i-1,j,k}}{l_x^2} + \frac{V'_{i,j+1,k} + V'_{i,j-1,k}}{l_y^2} + \frac{V'_{i,j,k+1} + V'_{i,j,k-1}}{l_z^2} - \left(\frac{2}{l_x^2} + \frac{2}{l_y^2} + \frac{2}{l_z^2} \right) V'_{i,j,k} \right) = V_{i,j,k}$$

Assuming $l_x = l_y = l_z = 1$, this simplifies to

$$V'_{i,j,k} = V_{i,j,k} + \nu\, \Delta t\, (V'_{i+1,j,k} + V'_{i-1,j,k} + V'_{i,j+1,k} + V'_{i,j-1,k} + V'_{i,j,k+1} + V'_{i,j,k-1} - 6\, V'_{i,j,k})$$

We also discretize the projection Poisson equation $\nabla^2 q(\vec{x}, t) = \nabla \cdot V(\vec{x}, t)$ for vector field V as

$$\frac{q_{i+1,j,k} + q_{i-1,j,k}}{l_x^2} + \frac{q_{i,j+1,k} + q_{i,j-1,k}}{l_y^2} + \frac{q_{i,j,k+1} + q_{i,j,k-1}}{l_z^2} - \left(\frac{2}{l_x^2} + \frac{2}{l_y^2} + \frac{2}{l_z^2} \right) q_{i,j,k} = \frac{V'_{i+1,j,k} - V'_{i-1,j,k}}{2\, l_x} + \frac{V''_{i,j+1,k} - V''_{i,j-1,k}}{2\, l_y} + \frac{V'''_{i,j,k+1} - V'''_{i,j,k-1}}{2\, l_z}$$

where $q_{i,j,k} \leftarrow q(\vec{x}, t)$, $V'_{i,j,k} \leftarrow V_x(\vec{x}, t)$, $V''_{i,j,k} \leftarrow V_y(\vec{x}, t)$, and $V'''_{i,j,k} \leftarrow V_z(\vec{x}, t)$. Assuming $l \leftarrow l_x = l_y = l_z$, this simplifies to

$$q_{i+1,j,k} + q_{i-1,j,k} + q_{i,j+1,k} + q_{i,j-1,k} + q_{i,j,k+1} + q_{i,j,k-1} - 6\, q_{i,j,k} = \frac{1}{2}\, (V'_{i+1,j,k} - V'_{i-1,j,k} + V''_{i,j+1,k} - V''_{i,j-1,k} + V'''_{i,j,k+1} - V'''_{i,j,k-1})$$

2.2 Implementation

[5] presents C code fragments of Stam's Stable Fluids algorithm as was just discussed. Accompanying this paper was the C source code of an interactive demo program implementing his algorithm using OpenGL's fixed shader pipeline. My goal for this component of the milestone was to reimplement the Stable Fluids algorithm in such a way that I could extend upon my progress in future milestones. Thus, I set the following specifications for my implementation work in this milestone:

1. The algorithm and its data should be encapsulated in class members for modularity.

2. Logical subroutines in the algorithm should be fully decomposed into computational primitives.

3. Grids should be abstracted as much as possible away from direct memory buffers, to allow for easy replacement of the actual class implementing the grid. However, there should be a fast way to access the memory buffers underlying grids so that they can be used as OpenGL textures.

4. The grids used by the algorithm should be represented as Arrays in Eigen. This may allow for the easy use, if appropriate, of linear systems solvers provided by Eigen. Milestone 2 requires extension of these data structures into three dimensions, which can be stored by Eigen's experimental (and officially unsupported) Tensor class.

5. The algorithm's ongoing state will be displayed in an interactive OpenGL application using the modern programmable shader pipeline, by rendering the fluid system as a texture. This will enable future modification of fluid rendering via fragment shaders, such as in the optional last milestone.

For my implementation, I made:

1. A VectorField class composed of multiple uniform two-dimensional grids (stored as Eigen Arrays) to couple all components of a vector field in arithmetic operations and function applications. Dye density is then a VectorField with one component (i.e. a scalar field), while fluid velocity is a VectorField with two components.

2. A FluidSystem class to hold the state of the simulation system as a dye density VectorField and a velocity VectorField, and to provide methods for simulating the evolution of the system.

I followed the LearnOpenGL's 2D Game Rendering tutorial series to implement classes for 2-D rendering in OpenGL's programmable pipeline. To this end, I made:

1. A Shader class (with minor modifications from the tutorial) to manage shader loading and uniform sharing.

2. A FluidTexture class (with heavy modifications from the tutorial's Texture class) to manage texturization of FluidSystems.

3. A ResourceManager class (with moderate modifications from the tutorial) to manage Shaders and FluidTextures.

4. A Canvas class (with moderate modifications from the tutorial) to manage how FluidTextures are drawn, along with the top-down camera's pose.

5. An Interface class (built on the structure from the tutorial) to respond to user interactions and integrate the Canvas, the ResourceManager, and the FluidSystem.

I then finished implementing the simulation system. From an initial condition of a square of dye with a small constant upwards velocity near the center of the grid, the system qualitatively appears to evolve very similarly to how it does in Stam's demo:

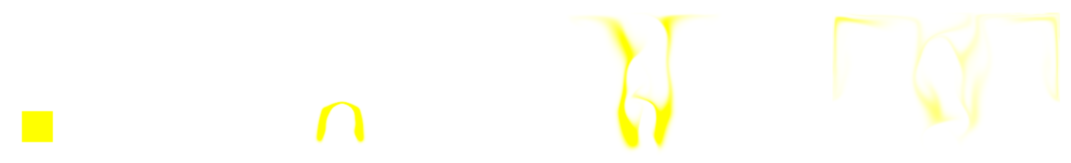

Figure 1. Deterministic evolution of the system from initial conditions, with diffusive (dye dissipation) and viscosity (velocity dissipation) constants set at 0. Screenshots taken of my OpenGL program. Note the turbulent flow in the center and the eventual laminar flow at grid boundaries.

However, the numerical dissipation that Stam's method suffers from is very evident, as can be seen in the rapid spread, blending, and fading of thin strands of food coloring. This justifies the work proposed for Milestone 3, namely implementation of Kim et al.'s BFECC extension of Stam's method to mitigate this numerical dissipation and produce more realistic results.

Additionally, the camera enables me to pan, zoom, and rotate on the canvas:

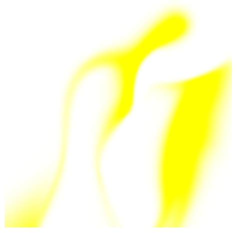

Figure 2. Canvas after panning, zooming, and rotating relative to the center of the viewport.

2.3 Preliminary Performance Profiling

On my low powered Chromebook (Intel Core i3 processor @ 1.70 GHz with integrated graphics) with full compiler optimizations (level `-O3`), each 300-by-300 grid takes an average of 91 ms to compute, which is approximately 10 fps. Each 200-by-200 grid takes an average of 40 ms to compute, each 400-by-400 grid takes an average of 166 ms to compute, and each 500-by-500 grid takes an average of 250 ms to compute. At this size scale, performance appears to scale linearly with the number of grid cells, as expected of a simulation which performs arithmetic operations on each cell of a grid based on a constant number of neighbors:

Grid Size (# cells)	Compute Time (ms/frame)
400	40
900	91
1600	166
2500	250
3600	500

Table 1. Size of grid versus time required to compute a frame of the grid. Each linear solve operation is run for exactly 20 steps, rather than to convergence. Linear least-squares fit gives $(\text{Compute Time}) = 0.001004 * (\text{Grid Size}) + 1.18$ with $R^2 = 0.999$.

Profiling with Callgrind reveals that the computational cost of the simulation is dominated (by an order of magnitude) by the linear solver, which currently runs 20 iterations of the Jacobi method. Interestingly, replacing the Jacobi method with the Gauss-Seidel method, which is known to converge slightly more quickly while also removing the need for a temporary additional data structure, resulted in much worse performance (160 ms to compute each frame of a 300-by-300 grid). This may be due to some quirks of compiler optimization or memory caching & access that make Jacobi iterations faster than Gauss-Seidel iterations. Regardless, this profiling result shows that the primary target of performance optimization should be the linear solver, and that the higher-level abstractions I used for grids (instead of manual indexing into buffers in memory) do not significantly hurt performance. Evaluating other methods of solving the linear systems in the Stable Fluids algorithm may be a useful objective for Milestone 3.

3 Milestone 2

This milestone consists of:

1. Allowing for simulation with dyes of different colors.
2. Extending the fluid system to 3 dimensions.

3.1 Colors

Milestone 1's results only simulated dye density as a scalar field, namely a VectorField with one component. Adding additional color channels was then a matter of specifying dye density as a 3-component VectorField for cyan, magenta, and yellow. I chose to use the CMY subtractive model of color blending rather than the RGB additive model to more closely model how food coloring mixes. Then I modified my FluidTexture class to store and track all color channels in the dye density VectorField. For clarity and simplicity, I chose to pass each color channel as a separate "red" texture to the fragment shader and blend the color channel textures in the shader, rather than reshaping the separate arrays underlying the color channels into a single RGB texture to pass into the fragment shader.

I had to reduce the system's grid size from 300-by-300 to 200-by-to-200 to maintain reasonable performance (currently 60 ms/frame). From an initial condition of differently-colored squares of dye with a moderate constant upwards velocity in the lower half of the grid, the system behaves as expected:

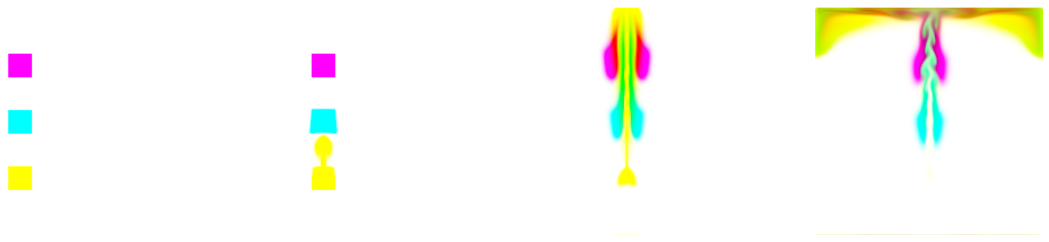

Figure 3. Note the subtractive color mixing, with the appearance of green and red. Note also the eventual evolution of the Karman vortex street.

3.2 Extension to Three Dimensions

To extend my system into three dimensions, I used Eigen's experimental (and unsupported) Tensor class for my grids and VectorFields rather than Eigen's Array class. Interestingly, doing this with a two-dimensional Tensor (as a direct replacement for an Array) provided a small performance boost to the software. While this was conceptually simple, I ended up spending a lot of time here debugging uninitialized Tensor elements. However, overall adding a third dimension significantly slowed down the simulation, making performance optimization crucial to achieve interactive usage.

After extensive preliminary experimentation, I found that the depth of the system should be quite small compared to the length and width of the system, and that velocity sources should be quite high compared to the length scales of the system, in order to begin to approximate the effect of soap in milk with dye.

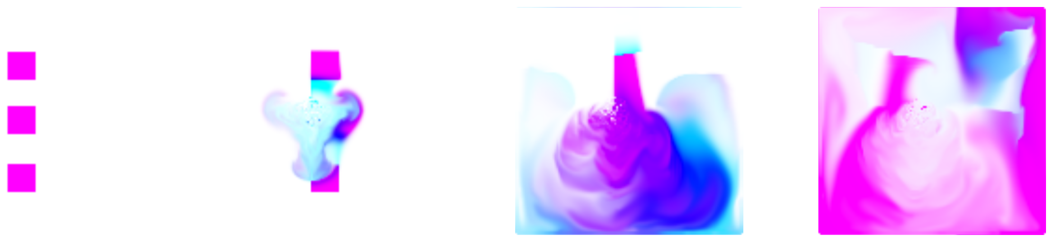

Figure 4. The constant velocity source is at the upper left corner of the central square in the initial conditions (leftmost image). Initial conditions also had dye in layers below the uppermost layer, though only the uppermost layer is rendered. This leads to evolution over time of the color of the fluid exiting the constant velocity source, as dye is transported underneath the surface and eventually pulled up by the constant velocity source or by other convection cells. Note the sharp edges visible in the rightmost image. These appear to be fluid separations from vertically-oriented convection cells, and are extremely unrealistic; during the simulation, these separations persist for quite a while. If fluid is flowing under the topmost layer, blending the top few layers of the fluid may solve this problem while also better approximating the appearance of milk. Note also the excessive diffusion of dye, despite a null diffusion constant; this is a consequence of numerical dissipation as discussed previously.

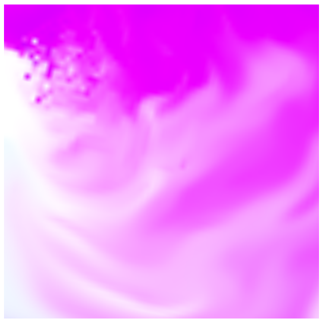

Figure 5. On left, screenshot of the simulation with adjusted camera pose. The specks of color in the upper left corner were unexpected and result from the extremely high fluid velocity in the area. While not strictly physically accurate, they resemble some of the pigment particles shed by solid dye sources placed in the milk-soap system, as seen in the reference video in figure 6. Additionally, the softer variation in color seen here resembles some scenes in that reference video.

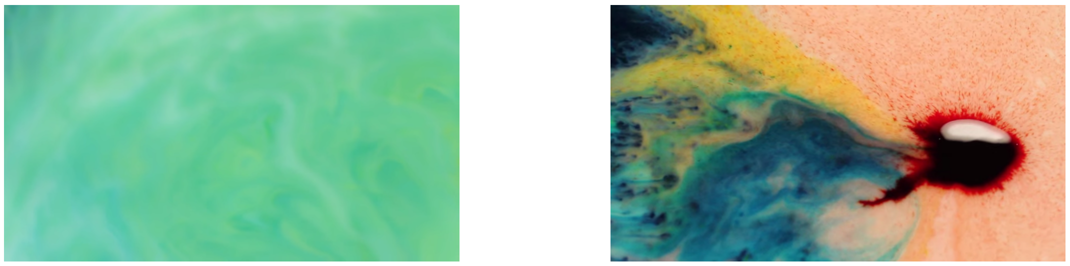

Figure 6. Screenshots from the visual reference video for this project [6], for comparison to my current progress. Achieving the level of detail seen in the right screenshot would require extreme performance improvements that would be beyond the scope of this project.

4 Summary

Having completed Milestones 1 and 2, I am ready to proceed with Milestone 3, namely improvement of simulation accuracy while maintaining reasonable performance. At a minimum, this will involve implementation of the BFECC extension to the Stable Fluids method [3] and use of a faster sparse linear solver (preferably a black-box solver) and/or a higher-order particle tracer.

Bibliography

[1] Dan Morris . Dan Morris's Notes on Stable Fluids (Jos Stam, SIGGRAPH 1999). Https://www.techhouse.org/~dmorris/projects/summaries/dmorris.stable_fluids.notes.pdf.
[2] Mark J Harris. *GPU Gems*, volume 1, chapter Fast Fluid Dynamics Simulation on the GPU. Addison-Wesley, 2004.
[3] ByungMoon Kim, Yingjie Liu, Ignacio Llamas, and Jarek Rossignac. Advections with Significantly Reduced Dissipation and Diffusion. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):135–144, January 2007.
[4] Jos Stam. Stable Fluids. In *Proceedings of the 26th annual conference on computer graphics and interactive techniques*, SIGGRAPH '99, pages 121–128. New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
[5] Jos Stam. Proceedings of the Game Developer Conference. March 2003.
[6] Michael Zoidis and Jodie Southgate. Olafur Arnalds & Nils Frahm - a2 (Official Video). YouTube Video, November 2012.