Isabelle Chong and José Cruz Mendoza
6.837 Final Project Report

Grid-Based Fluid Simulation in Two Dimensions

Motivation

Fluids can be quite difficult to procedurally animate, especially for grid-based designs, because they rely on an Eulerian rather than a Lagrangian model. Our goal for this project was to create a 2D grid-based fluid solver that would follow the Eulerian model to update the different values in its voxel grid, creating a stable, incompressible 2D fluid simulation.

Background

The approach to this project was based on *Practical Animation of Liquids* by Nick Foster and Ronald Fedkiw, *Fluid Simulation for Computer Graphics* by Robert Bridson, and *Real-Time Fluid Dynamics For Games* by Jos Stam. We also viewed some sample fluid solvers that are available on GitHub: one in Python by Alberto Santini and one in C++ by Ethan J. Li. These works all use an approach that is grid-based and Eulerian, using a Semi-Lagrangian method to update the grid during advection. We combined various concepts from these papers to ultimately create our fluid solver. The overall equation that the fluid solver attempts to model is the Navier-Stokes equation of fluid motion.

$$\frac{\delta u}{\delta t} = -u \cdot \nabla u + \nu \nabla \cdot \nabla u - \frac{1}{\rho} \nabla p + f \quad \text{(Eq 1)}$$

Stam rewrites this equation and adds an equation that accounts for changes in density

$$\frac{\delta u}{\delta t} = -(u \cdot \nabla) u + \nu \nabla^2 u + f \quad \text{(Eq 2)}$$

$$\frac{\delta \rho}{\delta t} = -(u \cdot \nabla) \rho + \kappa \nabla^2 \rho + S \quad \text{(Eq 3)}$$

In our approach we created C++ code that implements equations 2 and 3.

Approach

Compared to the solvers described in the resources listed above, we implemented a much simpler version in two dimensions. We coded this solver in C++ using GLOO and the base code provided by assignment 3. The algorithm we used was mostly based on Stam's paper, with the other papers used to provide further reference and general understanding. The general steps we followed to complete this solver were:

1. Create a voxel grid to contain all of the fluid
2. Implement the density equation
    a. Adding fluid (source) into grid
    b. Diffusion of the fluid in grid
    c. Advection of fluid through grid
3. Implement the velocity equation
    a. Adding forces to grid
    b. Diffusion of velocities in grid

   c. Advection of velocities through grid
   d. Projection of pressure in grid
4. Implement bounds and bounds checking
5. Render density map in GLOO

Voxel Grid

The method we chose uses an overall Eulerian approach rather than a Lagrangian one, meaning that rather than tracing the fluid particles within the simulation, we sample the necessary quantities on a grid structure. The grid holds information about density, previous density, velocity in the x direction (u), previous velocity in the x direction (u_prev), velocity in the y direction (v), and previous velocity in the y direction (v_prev). The model we follow defines the velocity at the center of each cell, with an extra layer of cells along the edge of our grid to account for boundary conditions (Figure 1). At these boundary locations, we simply set outgoing velocity components to 0. For simplicity, we decided to use this grid as opposed to the staggered MAC grid that is typically found in more advanced approaches.

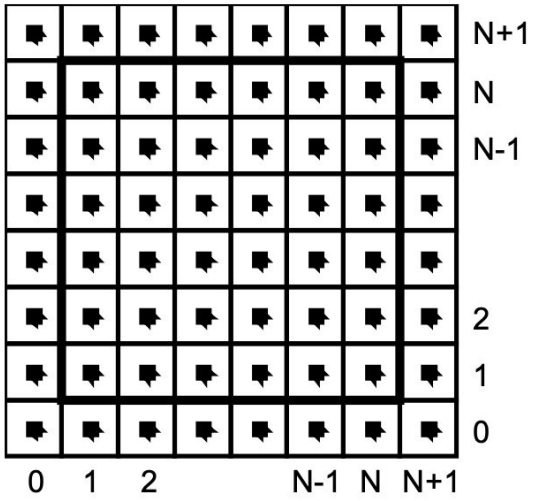


Figure 1: The voxel grid setup we used (Stam, 2003)

Density Equation

The density equation we decided to follow consisted of three elements: adding source densities, diffusion, and advection. For the sources, we use a vector containing density values at each location in the grid, for each color possible (C, M, or Y), to calculate the current density values using an Euler step. These source values are either pre-set or, in the steady state of the simulation, the previous density values.

We then want to diffuse this density, thus allowing densities from neighboring cells to contribute to each other. For this approach to remain stable, we use an implicit technique (backwards Euler), going back in time to find the density at a previous time step and interpolating within the grid at this previous time step to update the density value at that

particular point of the grid. To solve the system of equations that this approach creates, we use the iterative Gauss-Seidel relaxation as suggested by Stam.

Finally, to allow the density to be affected by the velocity within the voxel grid, we use self-advection, tracing each grid sampling location back in time by one time step using the velocity components u and v at that grid cell location. Once we have stepped back in time by one time step, we can bilinearly interpolate to find the values that were present in the previous time step. This technique is semi-Lagrangian since it treats the locations on the grid as "particles" that can move around in time.

These three different steps all combine into the following algorithm for updating the voxel grids' densities after one time step, *dt*, where *diff* is the rate of diffusion:

```
AddSource(density, density_prev, dt)
Diffuse(density_prev, density, diff, dt)
Advect(density, density_prev, u, v, dt)
```

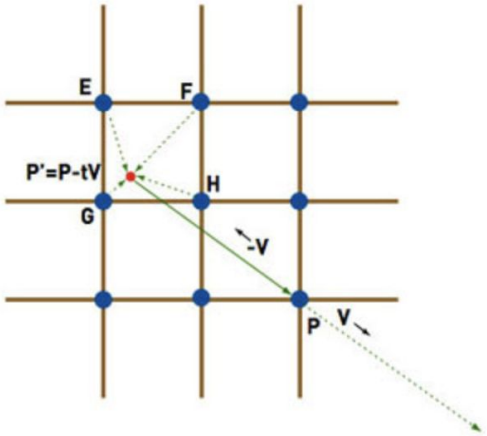


Figure 2: A visualization of self-advection on a point P in the grid
(from https://www.gamasutra.com/view/feature/1549/practical_fluid_dynamics_part_1.php?print=1)

Velocity Equation

The velocity equation follows similar steps to the density equation. Referenced papers added sources of velocity in the grid (such as gravity), diffused the velocity values according to the viscosity of the fluid, and then self-advected the velocity field. However, the velocity field also has to ensure that mass is conserved. In order to ensure that our field conserves mass, we use the Hodge decomposition, which states that every velocity field is composed of a mass-conserving field and a gradient field. By subtracting the gradient field from the velocity field, we can obtain a mass conserving field that contains the swirling geometry characteristic of fluids.

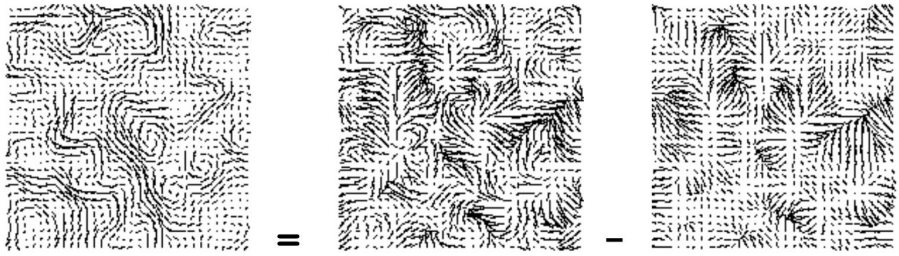


Figure 3: The mass conserving velocity field (Stam 2003)

In order to obtain this projection, we calculate the gradient from our velocity field by first solving a Poisson equation using the Gauss-Seidel relaxation previously used and then subtracting the gradient of the solution from our *u* and *v* velocity fields. Since advection is more accurate when the field is mass conserving, we call the *Project* method after diffusing the *u* and *v* velocities and after the advection step itself to ensure that the velocity field overall is mass conserving.

For the velocity step, we will combine forces, diffusion, advection, and projection after time step *dt* with viscosity *visc*:

```
AddSource(u, u_prev, dt)
AddSource(v, v_prev, dt)
Diffuse(u_prev, u, visc, dt)
Diffuse(v_prev, v, visc, dt)
Project(u_prev, v_prev, u, v)
Advect(u, u_prev, u_prev, v_prev)
Advect(v, v_prev, u_prev, v_prev)
Project(u, v, u_prev, v_prev)
```

Bounds

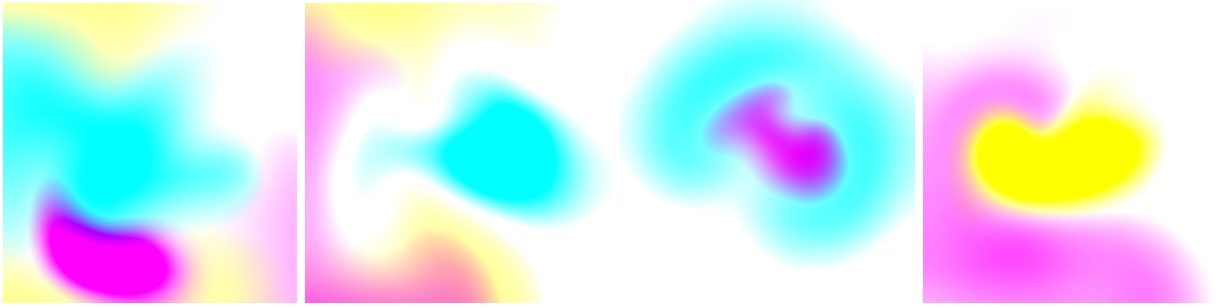
In order to make sure our fluid doesn't exceed the bounds of the canvas and to allow us to place objects into the canvas and have the fluid diffuse/advect around them, we incorporated a method to set bounds for the edges of the screen as well as for a list of rectangular object boundaries within the grid. The SetBounds method negates velocity in the u and v directions when encountering a boundary and zeroes out density. This allows us to see the effects from the *Results* section, especially those in figures 5 and 6.

Rendering

To visually represent our fluid, we created a plane VertexObject and shaded it using a custom shader that allowed us to update the color at each vertex. We then used this structuring to have the plane update its colors at each vertex after our system has processed a timestep, which allows us to see the fluid move around in the grid real-time. We also implemented methods that allow us to add densities, forces, and color changes. This was all done by using InputManager to process specific keys and interactions with the mouse and the canvas.

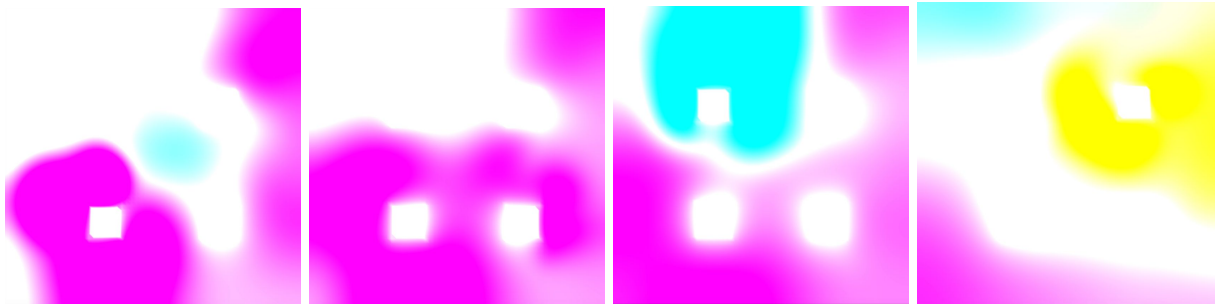
Results

Figures 4, 5, 6, and 7 show the results of our fluid simulator under different conditions: the addition of dye, interaction with boundaries, dam breaking, and interaction with wind forces.



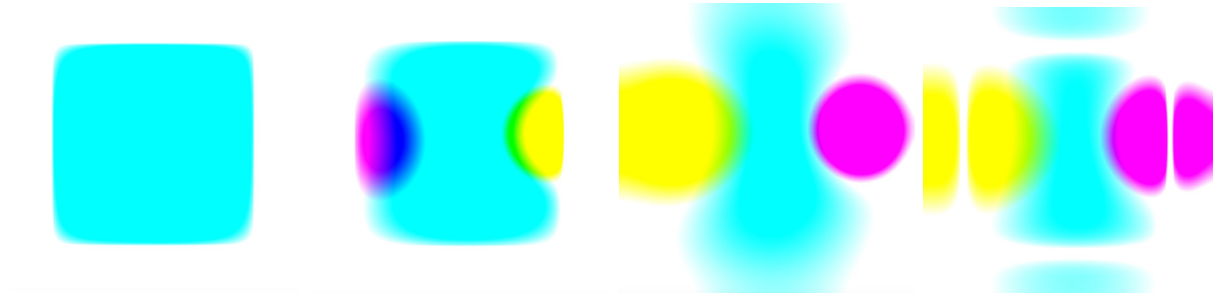
(a) (b) (c) (d)

Figure 4: Our 2D fluid simulation with different dye colors in motion



(a) (b) (c) (d)

Figure 5: Our 2D fluid simulation with the “OBJECT” setting turned on (boundaries in canvas)



(a) (b) (c) (d)

Figure 6: Simulation of the dam breaking model. In (a) and (b) we add a density source within the walls of the dam. Dam is removed in (c), allowing the dye to escape. In (d) we replace the dam, causing the cutting effect shown

(a) (b)

Figure 7: Simulation of our canvas with winds that cause the fluid to move in a whirlpool-like fashion.

Conclusion

For this project, we implemented a fluid simulator that simulates various colored fluids in a canvas in two dimensions. As was shown in the *Results* section, we incorporated (through the use of various user inputs) mixing of fluid colors, boundary detection, dam breaking, and wind. All of these features were built on top of the fluid simulation structure found in our referenced papers. Regarding possible extensions to this project, we believe that extending the coverage to three dimensions would be the next largest step in extending the solver that we currently have. This would entail increasing the resolution of our canvas and making sure that we would now be able to render colors/fluids in our density map inside of a three dimensional space (rather than the current plane that we display in our 2D solver). We could also mathematically extend our fluid solver to improve accuracy, using more accurate methods of modeling (such as the staggered grid), integration (such as the Runge-Kutta methods) and elimination (such as Jacobi iteration), as well as implementing vorticity confinement to prevent dissipation of our fluid dye.

Appendix

Video Links with Live Captured User Input

- Default and Object Boundary Model: https://youtu.be/gPU3y67FkMM
- Dam Breaking Model: https://youtu.be/lnt-czuSEaE
- Wind Model: https://youtu.be/XcR6pkNVc_s

Citations

Bridson, R. (2016). *Fluid Simulation for Computer Graphics*. CRC Press.

Foster, N., & Fedkiw, R. (2001). Practical Animation of Liquids. 10.21236/ada479067

Li, Ethan J. (2016). Simulating Dye Advection in a Three-Dimensional Fluid. https://ethanjli.github.io/project/cs-148-fluids/.

Santini, Alberto. (2018). Real-Time Fluid Dynamics for Games. https://github.com/albertosantini/python-fluid.

Stam, Jos. (2003). Real-Time Fluid Dynamics for Games.