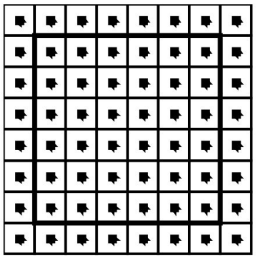
2D Fluid Simulation

Sec. 1 Overview

In general, there are two approaches to model the motion of fluid, the Lagrangian appraoch (particle-based) and the Eulerian approach (grid-based). In this project, I will focus on using Eulerian approach for 2D fluid simulation. Instead of tracking each particle of the fluid, we instead look at fixed points in space and see how the fluid quantities (such as density, velocity, pressure, etc.) measured at those points change in time. In other word, we dice up the entire 2D space into a grid and store the fluid quantity per grid:

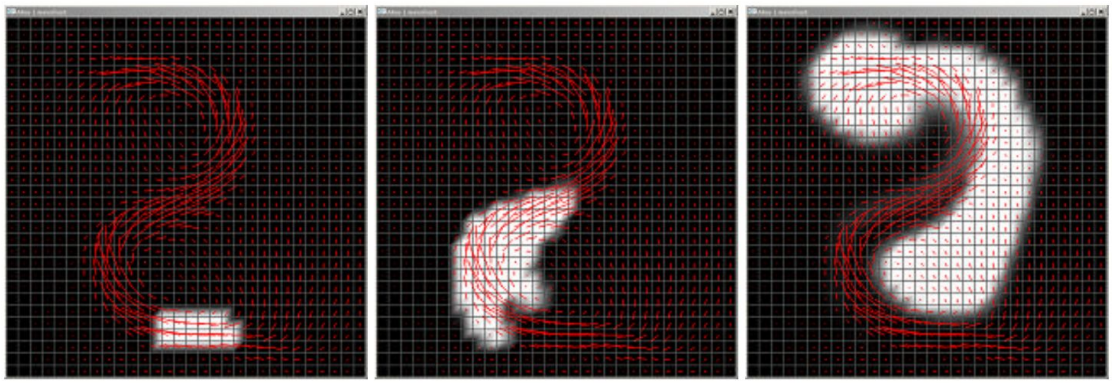


$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$

$$\nabla \cdot \vec{u} = 0$$

The fluid motion is governed by the famous incompressible Navier-Stokes equations, a set of partial differential equations that are supposed to hold throughout the fluid. They are usually written as above figure (right).

The Navier-Stokes equation required us to track both the velocity (u, which is a 2d vector) and the pressure at each grid center. We also have to track the number of fluid in each grid, which we will also call it as density (do not confuse it with fluid density, which is a constant used in Navier-Stokes equations). The “density” is used to track the motion of liquid and render in the screen space. Since the Navier-Stokes is too complicated to solve, we are going to split the motion of liquid motion into 3 steps: advection, body forces, and pressure/incompressibility. In the advection step, the fluid quantity (both the density and velocity) is moved through the velocity field as below:



Then the body forces like gravity is applied to the velocity field as in normal physical simulation. After the advection and body forces steps, the velocity of the fluid seldomly makes the fluid incompressible or respects the boundary condition (we assume there is a solid boundary), as a result, we need an extra step (called “project”) to make sure the fluid is incompressible. The “project” function will calculate and apply just the right pressure to make the velocity field u divergence-free, and also enforces the solid wall boundary conditions.
In summary, the complete algorithm for fluid simulation is as below (I strongly encourage the reader to read the book “Fluid Simulation for Computer Graphics” by Robert Bridson since my implementation is basically the same as theirs):

1. Initialize the density, velocity for each grid in the screen space. Here the velocity field is in into two scalar: u (in x direction) and v (in y direction).
2. During each timestep dt of the simulation:
    a. If the user decides to add new fluid, update the density in the user selected region
    b. advect(d, dt, u, v);
    c. advect(u, dt, u, v);
    d. advect(v, dt, u, v);
    e. v += dt * g; // g is the body force.
    f. project(dt, u, v);

I have supported following UI features:

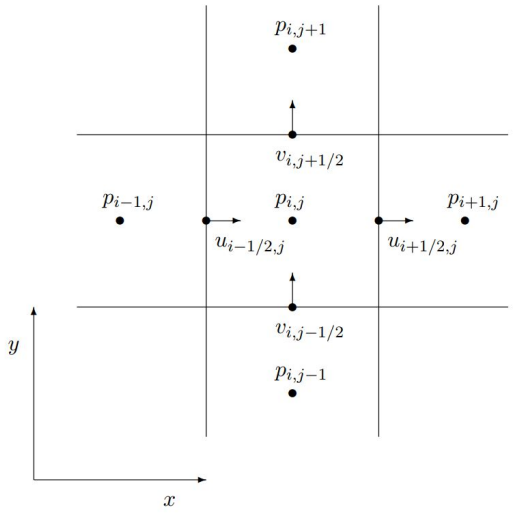
1. Add Fluid: The user can use mouse to draw a rectangle in the canvas, which will add some more fluid inflow to the canvas and they will move like fluid
2. Add Persist Fluid: You can click on “Persist Inflow” to make the fluid to be added every timestep, instead of adding it just once
3. You can change other parameters of simulation like “Preconditioned Conjugate Gradient (PCG) Iterations”, “Gravity”, etc.

Sec. 2 Implementation Details

In this section, I will focus on three parts: how I discretize the screen space into grids; how advect works and how project works. Most of the code related to fluid can be found in “FluidSolver.h”.

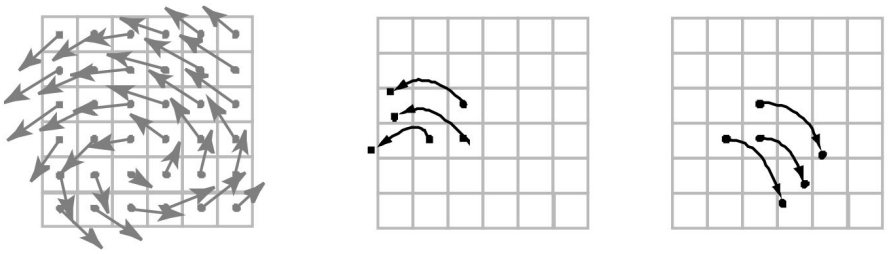
2.1 MAC Grids

The so-called “MAC grid” is a “staggered” grid, i.e. where the different variables are stored at different locations. The pressure and density in grid cell (i, j) is sampled at the centre of the cell.. The velocity is split into its x direction and y direction. The horizontal u component is sampled at the centers of the vertical cell faces. The vertical v component is sampled at the centers of the horizontal cell faces. As a result, the u/v is one column/row bigger than the density grid.



2.2 Advection

The “advection” forces the density and other quantities to follow a given velocity field. The key idea behind this is that moving densities would be easy to solve if the density were modeled as a set of particles. In this case we would simply have to trace the particles though the velocity field. For example, we could pretend that each grid cell’s center is a particle and trace it through the velocity field. The problem is that the center of a grid may move to any location (not necessarily at the center of a new grid). A better method is to find the particles which over a single time step end up exactly at the grid cell’s centers. In other words, we do a back advect for each “particle” and use linear interpolation to get the source density.



2.3 Projection

Now we'll look at making the fluid incompressible and simultaneously enforcing boundary conditions: the implementation of the routine project. The velocity in each grid is updated based on the pressure of neighbouring grids:

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x}$$
$$v_{i,j+1/2}^{n+1} = v_{i,j+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x}$$

Recall that the divergence of velocity field has to be zero in order to make fluid in compressible, which is:

$$\frac{u_{i+1/2,j}^{n+1} - u_{i-1/2,j}^{n+1}}{\Delta x} + \frac{v_{i,j+1/2}^{n+1} - v_{i,j-1/2}^{n+1}}{\Delta x} = 0$$

Combine above two formula together, we have a linear equation constraint for p which is:

$$\frac{\Delta t}{\rho} \left(\frac{4p_{i,j} - p_{i+1,j} - p_{i,j+1} - p_{i-1,j} - p_{i,j-1}}{\Delta x^2} \right) = - \left(\frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta x} \right)$$

Such constraint can be generated for each grid. We can tweak the above equation to model the solid boundary by replacing the velocity out of boundary as the velocity of solid object (0 for static solid wall) and plug in the pressure for solid boundary is below:

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t} \left(u_{i+1/2,j} - u_{\text{solid}} \right)$$

We have now defined a large system of linear equations for the unknown pressure values. We can conceptually think of it as a large coefficient matrix, A, times a vector consisting of all pressure unknowns, p, equal to a vector consisting of the divergences in each fluid grid cell, d (with appropriate modifications to divergence at solid wall boundaries): Ap = d. We can just invoke a linear system solver to get the correct pressure and then update velocity fields. Robert Bridson's book [1] introduced a very efficient representation for A and a preconditioned conjugate gradient algorithm for solving Ap = d in Chapter 4. I have implemented the PCG-based solver in my code. The pseudo code is at below:

- Set initial guess $p = 0$ and residual vector $r = d$ (If $r = 0$ then return p)
- Set auxiliary vector $z = \texttt{applyPreconditioner}(r)$, and search vector $s = z$
- $\sigma = \texttt{dotproduct}(z, r)$
- Loop until done (or maximum iterations exceeded):
  - Set auxiliary vector $z = \texttt{applyA}(s)$
  - $\alpha = \rho / \texttt{dotproduct}(z, s)$
  - Update $p+ = \alpha s$ and $r- = \alpha z$
  - If $\max |r| \leq \texttt{tol}$ then return p
  - Set auxiliary vector $z = \texttt{applyPreconditioner}(r)$
  - $\sigma_{\text{new}} = \texttt{dotproduct}(z, r)$
  - $\beta = \sigma_{\text{new}} / \rho$
  - Set search vector $s = z + \beta s$
  - $\sigma = \sigma_{\text{new}}$
- Return p (and report iteration limit exceeded)

Figure 4.1: The Preconditioned Conjugate Gradient (PCG) algorithm for solving $Ap = d$.

References

1. Robert Bridson - Fluid Simulation for Computer Graphics
2. David Cline et al. - Fluid Flow for the Rest of Us
3. Jos Stam - Real-Time Fluid Dynamics for Games