Animation & CGI Motion: Fluid Simulation

Theme 5 Milestone 2

Academic Honesty Policy

You are permitted and encouraged to discuss your work with other students. You may work out equations in writing on paper or a whiteboard. You are encouraged to use the discussion boards to converse with other students, the TA, and the instructor.

HOWEVER, you may NOT share source code or hardcopies of source code. Refrain from activities or the sharing of materials that could cause your source code to APPEAR TO BE similar to another student's enrolled in this or previous years. We will be monitoring source code for individuality. Source code should be yours and yours only. Do not cheat.

1 Introduction

In this theme, we will extend upon last week's theme and simulate *liquids*. In the milestone last week, our fluid was contained in a box, filling the entire box. There is a limit to the number of interesting things we can do in this environment except injecting dye and swirling it around. Liquid, as we know intuitively, is characterized by a *free surface*, i.e. the interface between the liquid and air. The motion of the free surface that creates many interesting phenomena. In this milestone we'll introduce a free surface to our simulation.

Additionally, we will introduce a way of handling arbitrary solid boundary conditions. What we implemented in the last milestone was not physically justifiable and only applies to simple boundary shapes like a box. This week we'll learn to handle solids of arbitrary shapes.

2 Inviscid Liquid

A specific but very prevalent type of fluid is the *inviscid liquid*. Liquids have a higher density than gasses, and thus will generate free surfaces due to gravity and other forces. Liquids come in many different types, yet here we will concern ourselves mainly with inviscid liquids. Being inviscid means simply that viscosity is not a dominant factor in the behavior. Many fluid phenomena we see in our everyday life fall into this category, including water. The following sections are devoted in properly handling the boundary conditions and free surface tracking, but before we dive into those, let's first make a convenient simplification: we can neglect the diffusion term in the Navier-Stokes equation, due to the low viscosity we'll be dealing with. Remember from last milestone that the semi-Lagrangian method is *unconditionally stable*, which is one of its selling points, but this comes at the price that it actually introduces extra damping in a form similar to a spurious viscosity effect, even if we don't model a viscosity term in the equation. This is termed *numerical dissipation* and it is an intrinsic property of this method. There are methods to reduce its effects, but for our purpose here, it is actually not a bad thing - now it is completely safe for us to drop the diffusion term without worrying at all, since numerical dissipation would give us the slight viscosity we'd need to explicitly model otherwise.

For this milestone, you don't need to call the diffuse() function as a result of dropping the diffusion term. However, you should keep in mind that we are taking advantage of the artificial viscosity from the numerical dissipation, which is not physical and not readily controllable by our animators.

3 Air Boundary Condition

At the free surface, the boundary condition is different. We don't constrain the advancing or receding of the liquid surface. The velocity is not constrained at all, because the air in the cell next to the liquid cell follows whatever motion the liquid has, without fighting back. From physics we know that this is not exactly true, because the air actually exerts pressure on the liquid. However in most cases the pressure from the air can be assumed to be constant, and its direction always points toward compression of the liquid, which is exactly cancelled by the incompressibility condition. As a result, pressure doesn't introduce any net acceleration or do any net work, allowing us to simply assume that pressure is zero in the air cells. Of course a more physical value would be some non-zero constant, but since only the derivatives of the pressure, not the pressure itself, ever appear in our equations, it doesn't make any difference. For example, if cell $(i+1, j)$ is an air cell, then the equation for fluid cell (i, j) becomes

$$\frac{p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{h^2} = \nabla \cdot \mathbf{u}$$

The tricky part then becomes the interpolation of the velocities. With solid boundary conditions we simply fix the velocities into/out of and inside solids to be zero and so there will always be valid velocity values to interpolate from. However on the free surface, we only fix pressure and leave velocity whatever it should be, thus the velocities in the air cells (which may be used in the interpolation during advection) don't have valid values because our pressure solve isn't enforced there. In general if the grid nodes carrying the quantities you are interpolating from don't lie inside the fluid, we should find the closest point from the fluid and use its value. For this milestone, we will do so using the following simple trick: after we've projected our velocity to be divergence free, we propagate the velocities from the fluid into the air cells by averaging the velocities of all fluid cells adjacent to an air cell and assigning the average to be the air cell's velocity. To be safe we should propagate into air cells by a few layers of cells (e.g. 5 layers), so as to make sure no interpolation will make use of the garbage velocity values originally in the air cells.

Let's make this process more sounded. In the first step of propagation, we detect which type the cell is, perhaps solid, fluid, or air. This can be done by checking our boolean-type array $m\_has\_fluid(i, j)$ and $m\_has\_solid(i, j)$, which will be one if the cell at (i, j) is fluid or solid. Here we also need to consider the MAC grid we are using. For example, for $u_{i,j}$ we check the cell at (i, j) and $(i, j+1)$, if they are both air-cell, we set $u_{i,j}$ to be ∞ (you may use SCALAR_INFINITY for this in your code) to mark it. Then for each $u_{i,j} = \infty$, we check its neighbor cells, namely the cell at $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$ and $(i, j - 1)$ (note you should CLAMP the coordinates into the valid region, specifically $[(0, 0), (N, N - 1)]$ for u.), add value from each of them if that cell is $NOT\ \infty$. Then we divide the sum by the number of added values. In other words, we average the values from cells that are not set to ∞. We iterate this process for 5 times, then for each $u_{i,j} = \infty$ (or $v_{i,j} = \infty$, a.k.a. the cell that is still set to ∞), we set their value to be zero.

4 Jacobi Solver

The problem with a Gauss-Seidel solver is order-dependency. The calculation of the result of the current cell depends on the updated result of the previous cell, but on the old value of the next cell. As a result, changing the order of calculation can give different results, if not converged. This can produce artifacts for fluids with complicated boundaries. In this milestone, we implement a Jacobi solver. Like the Gauss-Seidel solver, Jacobi's solver is also iterative. Nevertheless, it is independent of the order of calculation.

During each iteration, we use two buffers (usually called "Ping-pong" buffers) for pressure calculation, notated as p^0 and p^1. Taking the example where a cell is totally surrounded by liquid, we at first use p^0 as source and p^1 as destination, and we calculate

$$p^1_{i,j} = \frac{p^0_{i-1,j} + p^0_{i+1,j} + p^0_{i,j-1} + p^0_{i,j+1} - h^2 \nabla \cdot \mathbf{u}(i, j)}{4}$$

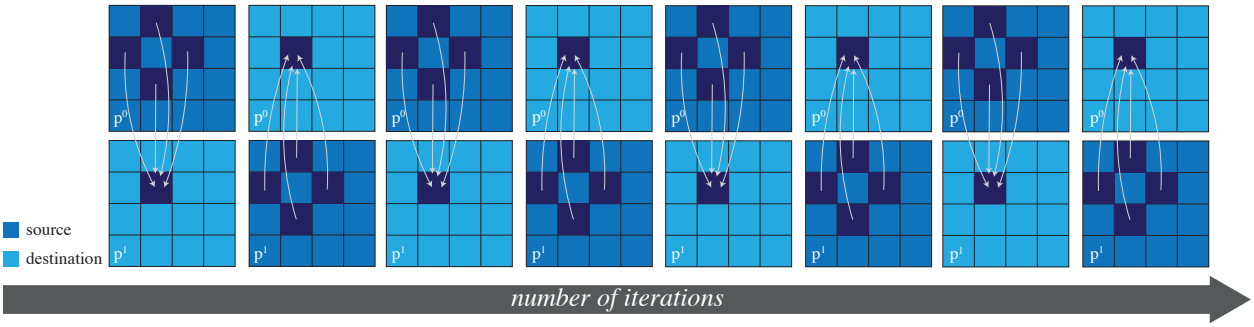

Figure 1: The Jacobi solver.

In the next iteration, we swap the source and destination and calculate

$$p^0_{i,j} = \frac{p^1_{i-1,j} + p^1_{i+1,j} + p^1_{i,j-1} + p^1_{i,j+1} - h^2 \nabla \cdot \mathbf{u}(i,j)}{4}$$

We continue swap p^0 and p^1 back-and-forth until we reach the maximal number of iterations (Fig. 1).

5 Free Surface Tracking

Having updated our simulator for boundary conditions, we still have to handle tracking the free surface. We know that the free surface boundary condition doesn't prescribe velocity values and thus the free surface can advance or recede with the velocity, meaning air cells can turn into liquid cells at the next time step, and vice versa. However, remember we assume that the free surface is perfectly aligned along cell walls, and every non-solid cell (which are decided from begining) is either a liquid cell or an air cell. This makes our life easier, because what we need to track is nothing more than a binary state (air vs liquid) for each cell.

We use a method called *Marker Particles* to track where the liquid is, i.e. which cells are liquid cells. Simply put, we keep a number of particles in the liquid cells, and advect them at every step along the velocity field to simulate the flow of liquid. These particles are merely initialized in the center of fluid cells, and then get evolved over time. Conceptually, these particles each represents a piece of liquid volume, and hopefully they will tell us where the liquid is at after being transported by the velocity. In the limit of infinite number of particles, this model becomes exact by modeling each individual liquid molecule explicitly, although in practice we typically use 4 (for 2D simulation) or 8 (for 3D) particles per grid cell. Once we have these particles, we can immediately determine which cells are liquid cells. We say a cell is a liquid cell whenever there is *at least one* particle in it, and a cell can only be an air cell when it contains no particles.

Note that this particle concept is different than the one we dealt with in the previous themes: usually particles are an indicator of a Lagrangian viewpoint but here we use particles on top of an Eulerian approach only for the purpose of liquid tracking (i.e. free surface tracking). These particles don't have mass or velocities, and they are passively transported by the velocity field defined on a grid.

What we need to do in the simulator is add a step after the velocity solve, to advect the particles along the newly computed divergence-free velocity field. For each particle we find out the velocity value at its position through interpolation (refer back to last week's Staggered Grid section for how to implement interpolation), and move the particle along the velocity.

6 Gravity and Volume Loss

One of the characteristics of the liquid we're trying to simulate is its interaction under a simple gravity force, which is the reason why the liquid stays at the bottom of its container and forms interesting patterns at its free surface. Liquid simulation in a space station looks completely different, and is dominated by surface tension, but we don't simulate that in this milestone. Therefore, we need to add gravity to our fluid simulator. Remember from class that the Navier-Stokes equation contains a body force ($\mathbf{f}_b$) term:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \Delta \mathbf{u} + \frac{1}{\rho}\nabla p + \frac{1}{\rho}\mathbf{f}_b$$

According to previous sections we ignore the viscosity term $\nu \Delta \mathbf{u}$, and apply the body force term $\frac{1}{\rho}\mathbf{f}_b$, the advection term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ and the pressure term $\frac{1}{\rho}\nabla p$ one after another, as detailed in Stam's paper. According to Newton's second law, the body force term is exactly gravitational acceleration:

$$\frac{1}{\rho}\mathbf{f}_b = \mathbf{g}$$

However, a new challenge arises once we've applied gravity this way: the resulting velocity field is uniformly downwards except at the bottom of the container, which has to have zero velocity. This yields a large negative velocity divergence at the bottom (meaning strong compression), and incompressibility will give a steady pressure gradient that cancels out the entire velocity field, making the whole liquid body sit still. Of course this is natural or otherwise we won't be able to enjoy coffee from any container. Looking carefully, the problem is that since we are using an iterative solver to solve for pressure, perfect convergence is impractical for any reasonable sized simulations, and numerical errors will accumulate over time, causing serious volume change as time goes on.

Our answer to this problem represents a general solution for accumulated deviations, which is to measure the total error relative to the desired values, and compensate for the error by applying a restoration force proportional to the error in magnitude. Assuming errors accumulate at a constant rate, there will always be a certain error level when the restoration force is strong enough to roughly cancel the deviation tendency, and thus make the solutions stable, within a certain distance away from the true solution. In the fluid simulation context, this boils down to manipulating the divergence computation, so that if the current volume is smaller than the original volume at the beginning of the simulation, the divergence is manually decreased by some amount proportional to the volume that's been lost, so that the pressure solved will not exactly remove all the divergence in the velocity, but allowing some amount of "diverging", creating the effect of inflating the volume towards the original volume. Accordingly, we increase the divergence to respond to situations where current volume is larger than the original volume.

So how do we measure the volume? Obviously we can't use the number of marker particles even though they are supposed to represent liquid volume, because the number or markers never changes. When we suffer a volume loss, what happens is that more particles get squeezed into one cell and each particle represents less volume. On the other hand, the number of liquid cells in the grid could be a good measure of volume. We can count the initial fluid cells and use it as a reference for the whole simulation. At each time step, we add $\eta(N_{current} - N_{initial})$ to each cell's discrete divergence computed from finite differencing on the velocity grid. Here η has no physical meaning and is tuned completely empirically, since this compensation for volume change isn't physical to start with. Using a large η will keep the equilibration position close to the true solution but will make the simulation less smooth as the fluid cell count jumps from one integer to the next. Using a small η will make the simulation smooth but tolerant to some amount of volume change. For this milestone $\eta = 0.01$ is a default value for a balance of smoothness and volume loss error.

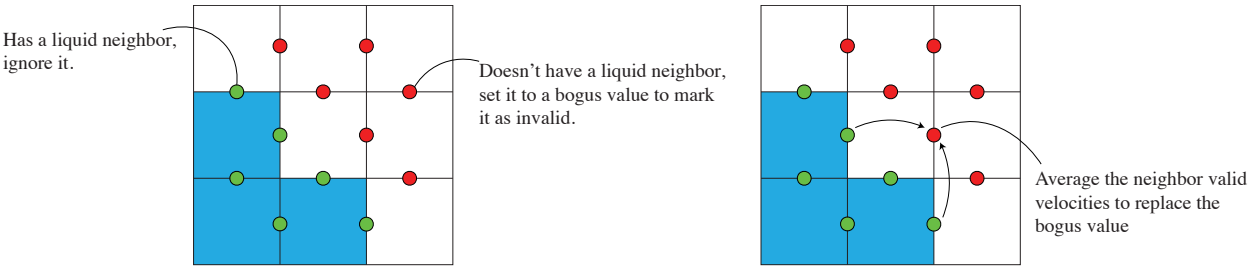

Figure 2: **Left:** mark all velocities with no neighboring liquid or solid as invalid; **Right:** update the invalid velocities by taking the average of the neighboring valid velocities.

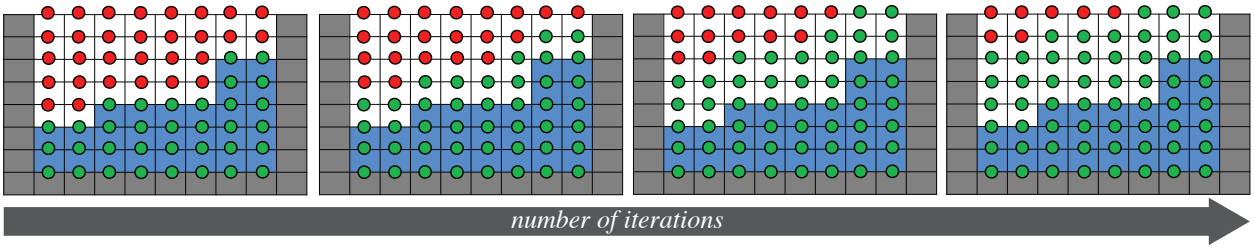

Figure 3: Average and grow outwards, layer by layer. Just treat each component separately. **Green Dots:** valid velocities. **Red Dots:** invalid velocities.

7 Velocity Extrapolation

So far we have solved for velocities touching a fluid cell. Nevertheless, we also need velocities in the air. The reason being that in advection step, we need interpolate at points that are not strictly inside the fluid.

The standard approach (as discussed above) is to use air velocity of the closest point on the surface. There are numerous ways to achieve this goal. For example, we can do a geometric search where needed; or we can propagate (extend) the velocities from the liquid into the air, by using either a fast marching method or solving a partial differential equation.

In this milestone we will consider a very simple iterative method that gives plausible results. We call it *velocity extrapolation*, which can be done as following:

1. Tag all the velocity samples set by the pressure solve to valid.
2. For a few iterations

   Set each invalid value having at least one valid neighbour to the average of its valid neighbours (Fig. 2)

   Next, tag each of the values you changed as valid

   Repeat.
3. set all remaining invalid velocities to zero.

A illustrative example over iterations is shown in Fig. 3.

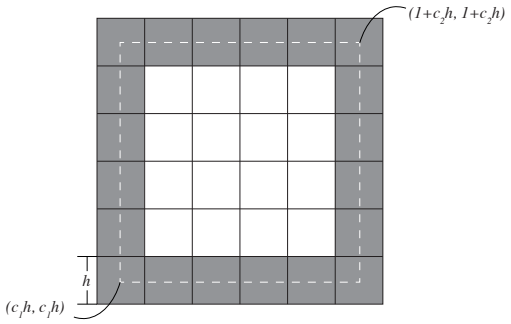

Figure 4: Clamp the advected particles into range.

8 Required Features

All implementation for this milestone should be done in the *StableFluidsSimWithMarkers* class.

8.1 Pressure Solve

Implement the StableFluidsSimWithMarkers::project() function, which should do the following:

1. set velocities on solid boundaries to zero
2. compute the discrete divergence of the velocity field (Sec. 3), with modifications to compensate for volume change (Sec. 6)
3. solve the Poisson equation with Jacobi method for pressure
4. apply the pressure gradient to the velocity field
5. set solid boundary conditions again to overwrite the wrong results coming from applying pressures above
6. propagate liquid velocities to nearby air cells, up to 5 layers (Sec. 7)

8.2 Particle Advection

Implement function StableFluidsSimWithMarkers::advect_particles(), moving particles forward along the velocity field. Don't forget to clamp the destination values so they don't move out of the liquid region (except moving into an air cell). Ideally we should prevent any particle from moving into a solid cell, but since we allow arbitrary solid cell distribution in the grid, this can get a bit tricky and thus is not required. You just need to make sure the particles stay within the box of fluid. A piece of pseudocode is given in Algorithm 1.

Algorithm 1 Particle Advection

1: **for all** $\boldsymbol{x}$ in particles **do**
2: interpolate to get velocity at position $\boldsymbol{x}$.
3: explicitly update the position using the interpolated velocity $\boldsymbol{x}$.
4: clamp the updated position into range from $(c_1 h, c_1 h)$ to $(1 + c_2 h, 1 + c_2 h)$ (see Fig. 4).
5: **end for**

In the starter code we have given the coefficients c_1 and c_2, please use those coefficients to match the oracle.

9 User-Interface Controls

In addition to the left/right dragging and 'r' key introduced from the last milestone, this time we have the following new controls:

1. 'f' key switches surface rendering modes. The default renders the free surface extracted using a marching cube algorithm and is good for visual intuition. The other mode renders individual particles and is good for debugging.

2. 'd' key toggles debugging info rendering on/off. It is turned off by default. Turning on debugging rendering in the particle rendering mode will display each individual particles and the velocity grid (red lines for v, green lines for u) display, which can aid your debugging significantly.

3. numer keys 1 to 9 applies a prescribed the velocity field of a certain pattern, so that you can have repeatability in debugging. Check the code in function *setPrescribedVelocity* in class *StableFluidsSim* to see what each number does.

10 Future Work

A lot of the aspects in this milestone can be extended, improved or refined. There have been a vast body of literature on fluid simulation and many interesting approaches have been proposed. For starter here's a list of possible directions for improvement:

1. Geometry of the free surface can obviously be improved. There are also many tricks to post-process the surface to remove noise, increase details etc [2]. More recently, researchers have shown that the free surface can be directly evolved to simulate complicated effects such as bubbles [5] and droplets [6], without the simulation of the interior.

2. 3D liquid are often rendered by path tracing for reflection, refraction and caustic effects. For complicated geometry, techniques such as bi-directional path tracing [11], Metropolis light transport [15], energy redistribution path tracing [4], or photon mapping [9] are used for effective rendering.

3. Liquid under smaller scale (e.g. millimeters) or small gravity (e.g. space station) is dominated by surface tension. Surface tension is typically modeled as a force that minimizes surface area or surface curvature. See [10, 14] for details.

4. Pressure solve is done with Jacobi method, which is just one of the many iterative methods for solving linear systems. Another popular method for solving linear systems in this context is *Conjugate Gradient* method, and its variant *Preconditioned Conjugate Gradient* method [13]. For large scale simulation, a multigrid scheme can be employed for parallelization and acceleration [12].

5. Free surface tracking can be done using an implicit surface function that is advected by the velocity field directly. This idea leads to a collection of methods called *Level Set* methods [8].

6. The Eulerian grid size can be modified dynamically during simulation, and adaptively so that areas with lots of details (e.g. near the surface) can be given a finer grid size and thus more computation quota [3]. High-order representations may also be used for detailed simulation with coarse grid [7].

There are also many other directions that it's impossible to list them all here. Interested readers are referred to Robert Bridson's book on *Fluid Simulation for Computer Graphics, 2nd edition* [1].

11 Creative Scene

There is no support for creative scenes for this milestone.

12 Grading

The oracle for this milestone is recently developed. Therefore if your code fails to match the oracle result and we find the failure is due to the oracle, your code will be graded *manually* based on if it appears visually correct, and if it runs interactively. With that said, we strongly urge you to start early and ask questions on the discussion board if you have any questions about your progress.

References

[1] Robert Bridson. *Fluid simulation for computer graphics*. CRC Press, 2015.

[2] Tyson Brochu. *Dynamic explicit surface meshes and applications*. PhD thesis, University of British Columbia, 2012.

[3] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics (TOG)*, 30(4):82, 2011.

[4] David Cline, Justin Talbot, and Parris Egbert. Energy redistribution path tracing. *ACM Transactions on Graphics (TOG)*, 24(3):1186–1195, 2005.

[5] Fang Da, Christopher Batty, Chris Wojtan, and Eitan Grinspun. Double bubbles sans toil and trouble: discrete circulation-preserving vortex sheets for soap films and foams. *ACM Transactions on Graphics (TOG)*, 34(4):149, 2015.

[6] Fang Da, David Hahn, Christopher Batty, Chris Wojtan, and Eitan Grinspun. Surface-only liquids. *ACM Transactions on Graphics (TOG)*, 35(4):78, 2016.

[7] Essex Edwards and Robert Bridson. Detailed water with coarse grids: combining surface meshes and adaptive discontinuous galerkin. *ACM Transactions on Graphics (TOG)*, 33(4):136, 2014.

[8] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational physics*, 183(1):83–116, 2002.

[9] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*, volume 364. Ak Peters Natick, 2001.

[10] Myungjoo Kang, Ronald P Fedkiw, and Xu-Dong Liu. A boundary condition capturing method for multiphase incompressible flow. *Journal of Scientific Computing*, 15(3):323–360, 2000.

[11] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. 1993.

[12] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 65–74. Eurographics Association, 2010.

[13] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[14] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk. A multiscale approach to mesh-based surface tension flows. In *ACM Transactions on Graphics (TOG)*, volume 29, page 48. ACM, 2010.

[15] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.