A Method of Fluid Simulation

EDWIN PAN, McGill University

Fluid simulations with boundaries between volumes filled with liquid and volumes without liquid can be achieved through Particle-In-Cell methods where the fluid presence is represented by units of fluid as particles within grid space whereby the lack of particle in a cell represents an air cell. Here, we re-implement an existing javascript implementation of Fluid-Implicit-Particle in C++ which simulates the behaviour of a liquid in a void.

Code: https://github.com/Tang-E/COMP559W23FP260843175
Video: https://youtu.be/csRBQIBRP5U

Additional Key Words and Phrases: FLIP, Fluid-Implicit-Particles, PIC, Particle-In-Cell, Fluids, Numerical Integration

ACM Reference Format:
Edwin Pan. 2023. A Method of Fluid Simulation. 1, 1 (April 2023), 4 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

As has been discussed in class, fluids can be simulated through Eulerian methods as well as Lagrangian methods where the former represents fluids as a constant volume of behaviour with immobile sampling points and the latter represents fluids by discrete particles that move in space. Brackbill and Ruppel [1986] present a Fluid-Implicit-Particle Particle-In-Cell (FLIP-PIC) method where both Eulerian and Lagrangian methods are combined by simulating fluid particles whose velocities are reflected and refined by an underlying fluid grid. Although this paper presents a method for simulating a liquid volume within a non-full-liquid volume, it does not represent a volume of liquid and air. As Müller [2022] points out, the implementation provided fails to represent two real world behaviours: (1) air bubbles in a liquid and (2) waves generated by air currents.

A single step in the FLIP fluid simulating process we implement according to [Müller 2022] comprises Simplecton-integrating the particles, resolving wall and obstacle collisions by particles, passing particle velocity data into the grid, computing particle density per grid cell, eliminating divergence in the grid by Gauss-Seidel with an offset for divergence for high particle density cells, and then finally transferring velocities back into particles. Of note in this final grid-to-particles transfer step is that a ratio can be set for whether resulting particle velocities is adjusted to reflect the grid's new state or adjusted to reflect the grid's change in state as the former induces viscocity in particles and the latter maintains the randomness in particles. These will be discussed further in the methodology section.

One additional step was optionally added to our simulator to attempt to simulate cohesion. This has not been successful, but we will still nevertheless discuss certain resulting behaviours and compare it to the as-provided code this project is based off of.

Author's address: Edwin Pan, McGill University, edwin.pan@mail.mcgill.ca.

2023. XXXX-XXXX/2023/4-ART $15.00
https://doi.org/0000001.0000001

2 RELATED WORK

The basis of this project is Müller [2022]'s 18th video in his Ten Minute Physics series available on Youtube and GitHub where he implements a FLIP-PIC simulation in around 1200 lines of javascript code. We copy his code and adapt it for use in our C++ environment; as such, much of the code provided to you in this paper is his. That said, certain pieces of OpenGL code did not work when converted to C++ and had to be remade. Unfortunately, When replicating his initial conditions, our code does not run as efficiently. This is likely a consequence of our replacement OpenGL code which places more work on the CPU than the GPU for particle location computations - code that was made because the particle shader was not working in our C++ adaptation.

Though Müller's series on Youtube and GitHub is an extremely valuable resource, he does not cite a source for his methods - perhaps due to how well known FLIP is (he mentions its extensive use in the Movie industry). The earliest paper I could find on the FLIP method for simulating fluids is Brackbill and Ruppel [1986]. Given that it's from 1986 and is later further discussed in Brackbill et al. [1988], I assume the 1986 paper is the earliest work on the topic.

3 METHODS

We overview the numerical methods used to simulate the fluid provided to us by [Müller 2022] and his code - if you are familiar with his code or FLIP-PIC simulations, you can skip over section 3.1. After that, we will go over the code I added and the behaviour I was attempting to imitate.

3.1 Müller's FLIP-PIC Fluid Simulator Code

[Müller 2022]'s simulate() function calls, in order, for the:

(1) Integration of Particles
(2) Pushing Apart of Particles
(3) Particle Collision Resolving
(4) Particle-to-Grid Velocity Transferring
(5) Particle Density Updating
(6) Incompressibility Resolving
(7) Grid-to-Particle Velocity Transferring

The following subsections discuss each of these calls.

3.1.1 Integration of Particles. All particles are simplecton integrated such that their velocities are updated based on gravity and their positions are updated based on the updated velocities.

3.1.2 Pushing Apart of Particles. The algorithm looks at the new positions of each particle and checks whether they are colliding. If two particles are found to be colliding, then their distance is set such that they are exactly two particle radii apart. Note that [Müller 2022] makes use of a filtering algorithm to make collision detection highly efficient - but that we do not discuss this in this paper.

3.1.3 Particle Collision Resolving. The algorithm now looks for collisions between particle and obstacle as well as particle and wall.

The algorithm first tests for collisions between particle and obstacle before then checking for particle and solid. Particles found to be within particle radius + obstacle radius of the obstacle have their velocities set to that of the obstacle - allowing for the pushing and pulling of particles in the simulation. Then, particles found to be colliding with walls then have their x or y coordinate set to that of the wall and their x or y velocity set to 0 to the wall.

3.1.4 Particle-to-Grid Velocity Transferring. The algorithm now looks to inform the Eulerian velocities of the grid by the velocities of the particles. Velocities on the grid are placed on the midpoints of edges of the Eulerian grid. Each new grid velocity is calculated as a weighted sum of adjacent particles' weighted contributions of their velocities. Any particle's contribution to a particular grid's new velocity is determined through bilinear interpolation as each particle must distribute their velocity among the 4 grid velocity midpoints adjacent to it. Note that the simulator keeps a copy of the grid velocities before they were changed by the particles so as to be able to keep track of the change of velocity on the grid.

3.1.5 Particle Density Updating. The algorithm now takes count of the amount of particles in each grid cell. This is used to colour particles as well as to colour the grid UI of the simulator (high-density cells show up in red). It is also used to inform the next step where compressibility of the system is resolved.

3.1.6 Incompressibility Resolving. This section of the algorithm achieves two things: (1) it eliminates divergence in the system and (2) it resolves fluid compression. Divergence is solved as one usually would in an Euler fluid simulation: fluid velocities at each cell are balanced such that their outputs are 0 through Gauss-Seidel iterations. Compression of fluids is solved by adjusting the target output of each Gauss-Seidel iteration such that cells that have higher than desired particle counts have fluid velocities leaving the cell.

3.1.7 Grid-to-Particle Velocity Transferring. The final section of [Müller 2022]'s code involves transferring velocities back from the grid and into the particles. Transferring has to be done carefully in consideration of the resulting viscocity of the liquid in the simulation as simply replacing each particle's velocity with a bilinear interpolation of the 4 adjacent grid velocity points results in particles that all follow the same currents - eliminating normal effects of fluids like splashes or separating columns of water. To resolve this, particle velocities are instead get the change of grid velocity added to their velocities by bilinear interpolation. [Müller 2022] helpfully provides a flipRatio to determine whether the code will update particle velocities based on new grid velocities, based on change of grid velocities, or somewhere in between.

3.2 Trying to Simulate Cohesion

Very unscientifically, I attempt to add cohesion into the simulation. A diagram on Wikipedia illustrates cohesive forces on a liquid as a pulling force on any particle in a fluid from all other particles in the fluid such that particles on the boundary of a volume of liquid are pulled inward of the liquid due to lack of particle outside the liquid pulling it away [Wikipedia 2023]. To obtain this behaviour, we calculate a pulling acceleration between each particle in the following form:

$$c = -ln(epsilon)/maxDistance$$

$$pullingAcceleration = maxAcceleration * exp(-c * distance)$$

These equations were made such that calculated pulling forces would never exceed maxAcceleration (prevent the simulation from exploding) as well as ensuring that pulling forces would become insignifcant (epsilon, set to 1e-6 in code) at maxDistance. To be clear, this simulation is not at all meant to represent reality, but rather simply to attempt to imitate a behaviour of reality. But as we will discuss in the results section,

Using the acceleration function above, for all particles i, the net cohesion acceleration on i is the sum of all accelerations calculated above toward all other particles j. While it could perhaps be realistic to calculate this for all particles, this would be an $O(n^2)$ algorithm and slow our seconds-per-frame algorithm to a seconds-per-frame algorithm. We therefore partition the grid into square cells of width maxDistance such that the net cohesion acceleration on any particle i is only calculated based on particles j from cells in that of i and adjacent to i.

Even with this optimization above, however, the algorithm still starts running slow at around 1 second per frame. So I've added an option to try to make cohesion force calculation faster in the algorithm by aggregating particles over each cell such that cohesive force is calculated as the sum of the pullingAcceleration of a mean particle for a neighbouring cell j multiplied by the number of cells in that cell - and this is done for all 8 adjacent cells to particle i as well as to cell particle i itself is located in. This results in very strange behaviour as we will discuss in the results section, but at least the algorithm runs relatively fast.

However the cohesion acceleration is calculated on a particle, the acceleration is integrated over time period ∂t into the particle's velocity.

Sidenote: regarding my citation of Wikipedia: while we can definitely talk about how Wikipedia is a primary source for implementing physics properties, I would point out that the force calculation I pulled out of pure intuition is a bigger concern over the effectiveness of this algorithm. Because I don't think it does a good job of what I want.

4 RESULTS

When running the algorithm as [Müller 2022] wrote it, the algorithm runs well and the resulting fluid simulation resembles that of real liquids, albeit with motion slowed that more resembles the sea th an it does a fish tank from a real-time viewing standpoint. We run at the default 90% FLIP/PIC ratio which gives us behaviour that is not too viscous, making it resemble water as the fluid swishes from left to right and splahes on and from the walls as seen in Figure 1.

The simulation is also able to demonstrate lasting vortices, which is a desired effect of using particles to simulate fluids [Brackbill et al. 1988], as seen in Figure 2.

Due to how slow the particle-to-every-particle cohesion acceleration algorithm takes to run, I have not been able to examine its impact on the simulation. The effects of using the fast cohesion simulation however are very obvious when the max cohesion force

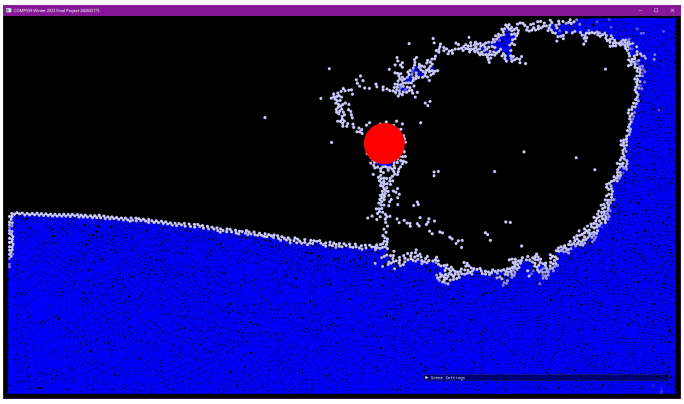

Fig. 1. Normal Simulation Settings

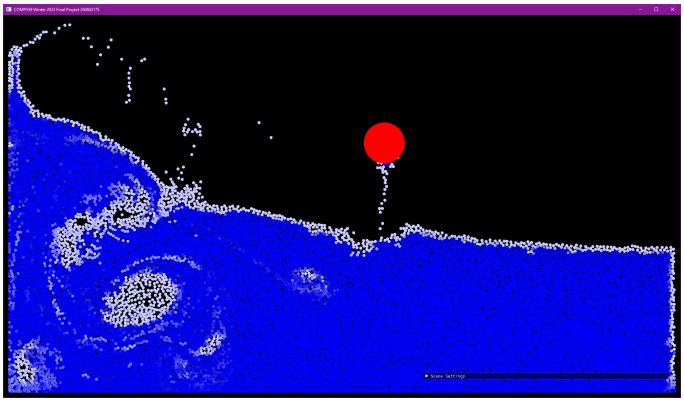

Fig. 2. Vortices in Normal Simulation

is set to a high 1 $1m/s^2$ acceleration and to very large squares as the simulation starts to exhibit bubbling across the centre of the grids as seen in Figure 3 which demonstrates shows the simulation just a handful of frames after a simulation restart. This bubbling persists throughout the simulation and makes the simulation more comparable to boiling water as seen in Figures 4 and 5, which is obviously not the intended behaviour.

5 CONCLUSIONS

[Müller 2022]'s code works effectively as producing fluid like behaviours that look natural, albeit without the ability to simulate bubbles and wind-generated waves. Further, I should probably look at actual cohesion force equations in the future before attempting to code that again (and it will certainly help to learn how to work better filtering algorithms to optimize performance).

6 EXTRA NOTES

I think next time I adapt another person's work that already works I'll just use their working code exactly and then make modifications within it - you know, avoid reinventing the wheel. While it was cool and all to be able to deeper understand [Müller 2022]'s code by actively rewriting his work, it was also a massive pain to have to

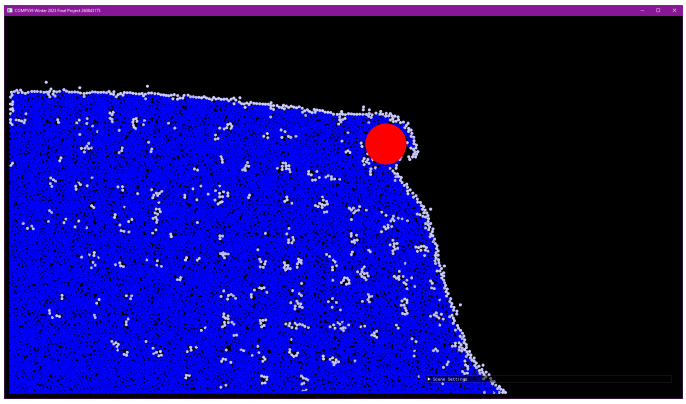

Fig. 3. Excessive Fast-Cohesion Computations causing grid-aligned bubbling

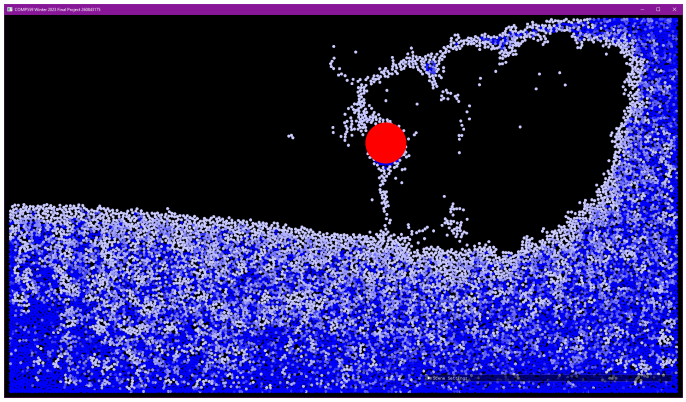

Fig. 4. Excessive Fast-Cohesion Computations resulting in bubbles permeating the simulation

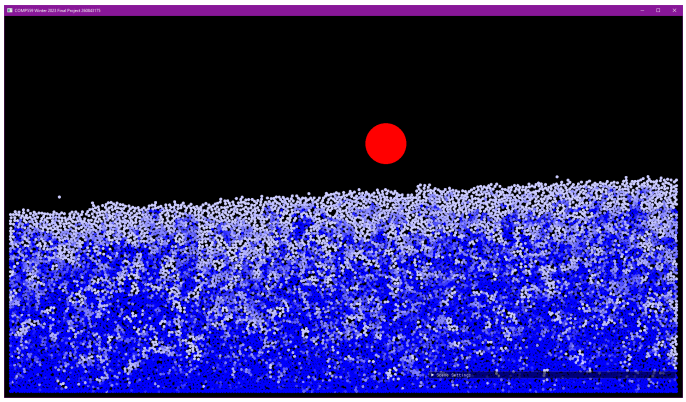

Fig. 5. Excessive Fast-Cohesion Computations still bubbling after water settles

set up a new coding environment and debug all the problems that occurred when adapting his javascript code in C++. I'd say 80% of the effort that went into this project was just getting his code to

work in C++ - which is a shame because it would have been really cool to have had some more time to really poke around in this code and see what happens when I add one new idea and another idea into the code. Had I known it would have taken me that long to get his simulator back up and running in C++ I think I would have just instead learned to code in WebGL javascript instead.

Anyway, I'm writing this section mostly to indicate that while there really is very little original contribution by me in this project that the mere process of adapting his code to this project was a large undertaking considering the scope of this project. But alas, more pain does not necessarily equate to a better paper. It was a lot of fun to get the fluid stuff working though - just that, if I'm trying to present something new, then I have not achieved anything here. Anyway judge it how you will. Thanks for reading.

REFERENCES

J. Brackbill, D. Kothe, and H. Ruppel. 1988. Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications* 48, 1 (1988), 25–38. https://doi.org/10.1016/0010-4655(88)90020-3

J. Brackbill and H. Ruppel. 1986. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys.* 65, 2 (1986), 314–343. https://doi.org/10.1016/0021-9991(86)90211-1

M. Müller. 2022. Ten Minute Physics. (2022). https://github.com/matthias-research/pages/blob/master/tenMinutePhysics/18-flip.html

Wikipedia. 2023. Surface tension — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Surface%20tension&oldid=1149654649. (2023). [Online; accessed 21-April-2023].