Real-time PIC/FLIP Fluid Simulation Using OpenGL Compute Shaders

Logan Zartman
logan.zartman@utexas.edu
Source code

May 19, 2020

1 Introduction

This project is an implementation of Particle-in-Cell/Fluid-Implicit-Particle (PIC/FLIP) fluid simulation, a hybrid Eulerian-Lagrangian technique for simulating fluid flows. I present an implementation designed for real-time simulation and visualization of 3D fluids, executed entirely on a GPU using OpenGL 4.3 compute shaders. This enables high performance simulation both by exploiting the parallel architecture of the GPU, and by avoiding copying between host and GPU memory for simulation or rendering. I use OpenGL 4.3 both for my familiarity, and because it is more widely accessible than some compute solutions such as CUDA. Further, I present an implementation of Screen-Space Fluid Rendering, which renders an approximation of the fluid surface without performing any reconstruction into triangles or other geometry. This allows for real-time effects such as reflection and refraction on the fluid surface. Ultimately, the project seeks to be a proof of concept for real-time, interactive fluid for applications such as games and digital media.

1.1 Background

The Particle-in-Cell method for fluid simulation was first published by Harlow and Welch in 1965 [5]. Harlow and Welch proposed a fluid simulation that performs most steps, the most important of these being pressure computation, on a discretized grid covering the simulation domain. In contrast to purely Eulerian (grid-based) fluid simulations, their technique performs advection (movement of fluid quantities through the fluid velocity field) using a set of discrete particles which lie on top of the grid. Quantities on the grid are transferred by interpolation to nearby particles to perform advection, and then particle quantities are distributed back to nearby grid cells to continue the next timestep of the simulation.

A serious issue with the PIC method is numerical dissipation. Since quantities are interpolated from grid to particles and back to grid, energy in the simulation is lost quickly. Additionally, detailed motion such as vorticity is quickly blended away. This problem is solved by the Fluid-Implicit-Particle method, proposed by Brackbill and Ruppel in 1965 [1]. In essence, this method reduces numerical dissipation by transferring *changes* in fluid velocities from the grid to the particles, rather than transferring the velocities themselves. However, this method can result in an overly-energetic simulation. PIC/FLIP attempts to solve both problems by blending the updated velocities produced by PIC with those produced by FLIP.

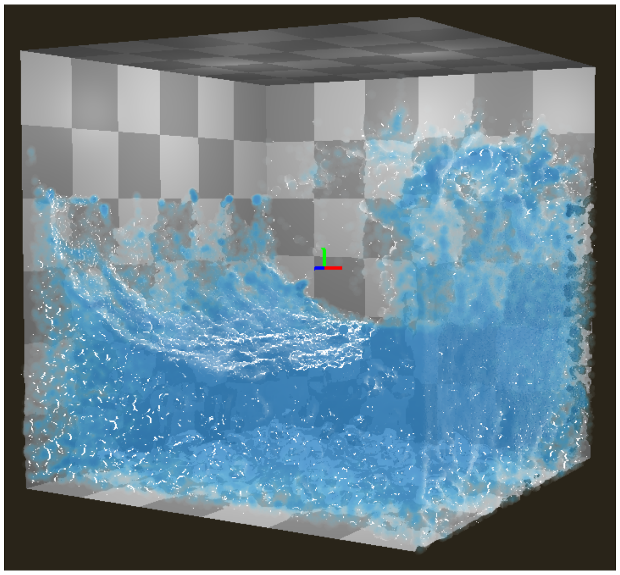

Figure 1: Simulation on a 24x24 grid with Screen-Space Fluid Rendering.

2 PIC/FLIP Method

I present a PIC/FLIP implementation based primarily on *Fluid Simulation for Computer Graphics* by Bridson [2]. I have made some choices which differ from those suggested by Bridson to support straightforward implementation on the GPU and, more specifically, using OpenGL APIs. Below, I describe each step in the PIC/FLIP simulation. I describe any interesting implementation decisions in the relevant section.

First, it is important to discuss a detail of the grid. Most PIC/FLIP implementations use what is called a Marker-and-Cell (MAC) grid, in reference to the grid structure proposed for the MAC method of fluid simulation. This grid samples fluid quantities other than velocity (such as pressure) in the center of the grid cell. Each component of velocity is sampled on the cell faces, such that each cell has two velocity components surrounding its center in each direction. This enables accurate, unbiased finite-difference estimations of derivatives. I recommend consulting the Bridson book for details.

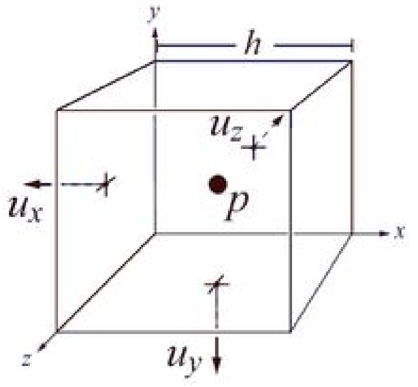

Figure 2: A MAC grid cell. Velocities u_x, u_y, and u_z are sampled on cell faces, while pressure p is sampled in the cell center. Image credit: Dombroski et al [3]

2.1 Particle-to-Grid Transfer

In this step, particle velocities must be distributed to the nearest 8 velocity elements on the grid. Since each velocity component is stored in a separate grid offset from the main grid, this distribution must be performed separately (choosing separate elements of each velocity grid) for each of the x, y, and z velocities. The fraction of a particle velocity distributed to a particular velocity element on the grid is based on how far away the particle is from that velocity element.

This step is nontrivial to perform on the GPU. In a CPU-based implementation, it is straightforward to iterate over each particle, locate the nearest grid corners, and accumulate velocities to these grid corners. One can parallelize this operation by transferring multiple particles in parallel. It is necessary to use atomic operations to accumulate velocities, since multiple particles may contribute velocity to one velocity element on the grid. However, this presents problems on the GPU. Firstly, GPU memory is by default incoherent–meaning that multiple threads cannot see each other's memory writes. Therefore, I not only use GLSL's `atomicAdd()` function, but I do this within a *work group* of large size (i.e. 1024 threads.) Threads within a work group can access shared, coherent memory. However, the size of a work group is fixed at compile time, so my particle-to-grid transfer shader is run multiple times, transferring a fixed-size batch of particles each time. A further complication is that `atomicAdd()` only supports integers, so floating-point velocities must be converted to and from fixed-point integer representations to achieve this transfer.

2.2 Body Forces

Body forces are forces that are not coupled with the fluid, such as gravity and wind. These forces are applied to the grid velocities by way of simple forward-Euler time integration: $v_1 = v_0 + \Delta t \cdot g$, where g is the acceleration on the fluid at a particular grid cell, Δt is the timestep, and v_1 is the updated velocity.

I implement body forces by dispatching one thread per grid cell and performing time integration.

2.3 Pressure-Solve Setup

The crux of incompressible fluid simulation is ensuring zero divergence, i.e. that the quantity of fluid entering a grid cell is equal to the quantity of fluid exiting it. This is equivalent to ensuring that the fluid is incompressible. Pressure is a quantity that is computed to enforce this condition in the velocity update (Section 2.5.) In the setup phase, I compute the divergence of velocity at each grid cell by computing the sum of finite differences of velocities:

$$(\nabla \cdot \overrightarrow{u})_{x,y,z} = \frac{u_{x+\frac{1}{2},y,z} - u_{x-\frac{1}{2},y,z}}{\Delta x} \tag{1}$$

$$+ \frac{v_{x,y+\frac{1}{2},z} - v_{x,y-\frac{1}{2},z}}{\Delta x} + \frac{w_{x,y,z+\frac{1}{2}} - w_{x,y,z-\frac{1}{2}}}{\Delta x} \tag{2}$$

Generally, u, v, and w are the x, y, and z components of velocity, and the half-offsets represent the positive and negative grid faces, respectively. Δx is the size of a grid cell.

In this step, I also compute coefficients for the matrix equation that is solved for pressure in the following section 2.4. The matrix represents a set of linear equations of pressure and divergence for each grid cell, derived by Bridson as the following:

$$\frac{\Delta t}{\rho}\left(\frac{6\mathrm{p}_{i,j,k} - p_{i+1,j,k} - p_{i,j+1,k} - p_{i,j,k+1} - p_{i-1,j,k} - p_{i,j1,k} - p_{i,j,k-1}}{\Delta x}\right) = -(\nabla \cdot \vec{u})_{x,y,z} \quad (3)$$

This equation assumes a grid cell at x, y, z that is surrounded on all sides by fluid-containing cells. For cells with no fluid (air cells), the pressure is set to zero (and removed from the equation.) The pressure coefficients are encoded as a sparse matrix, as proposed by Bridson. Solving for pressure in this set of one equation per grid cell gives pressure values that remove divergence from the velocity field in the velocity update in Section 2.5.

Both of the computations in this section are implemented by dispatching one thread per grid cell, and performing the aforementioned calculations.

2.4 Pressure Solve

This step solves the linear equation $A\mathbf{p} = \mathbf{b}$ for the cell pressures $\mathbf{p}$, where A is a coefficient matrix for the equations described in Equation 3, and $\mathbf{b}$ is the negative divergence, as computed in Equation 1.

I implement this step using a relatively simple iterative algorithm: the Jacobi Method. The Jacobi Method is a good choice for GPU implementation, because the solution guess can be updated in parallel for all of its elements. This, combined with the fact that it is straightforward to implement, led me to choose the Jacobi Method. The iterative update to the solution $\mathbf{p}$ is defined below:

$$A = D + L + U \quad (4)$$

$$\mathbf{p}^{k+1} = D^{-1}(\mathbf{b} - (L + U)\mathbf{p}^{k}) \quad (5)$$

In these equations, D is the diagonal matrix, L is the lower triangular, U is the upper triangular, $\mathbf{p}$ is the vector of pressure for each cell, k is the iteration index, and $\mathbf{b}$ is the negative divergence (as explained in Section 2.3.)

I implement Jacobi iteration by, for each iteration, dispatching one thread per grid cell to compute the updated pressure, and then dispatching one thread per grid cell to copy the updated pressure back to the pressure guess, to be used in the next iteration.

2.5 Velocity Update

The velocity update simply computes the derivative of pressure at each velocity element (for each grid of velocity components) by computing the finite difference approximation using the pressure values on either side of the velocity element. Since the MAC grid is staggered with pressures halfway between each velocity element, this calculation is simple and accurate.

I implement the velocity update by dispatching one thread per grid cell and updating the velocities in parallel.

2.6 Grid-to-Particle Transfer

To perform advection using the particles, the updated, divergence-free velocity field must be transferred from the grid back to the particles. For any particular particle, this entails finding, for each of the three velocity components, the nearest eight velocity elements on the grid. Then, the grid velocities are trilinearly interpolated into each component of the particle velocity, based on the distance of the particle from each velocity element.

To implement the PIC/FLIP update, I compute for each particle both the interpolated velocity and the interpolated *change* in velocity from the grid. I compute a FLIP-updated particle velocity by adding the change in velocity from the grid to the existing particle velocity. Finally, I interpolate between the interpolated grid velocity and the FLIP-updated particle velocity, using an arbitrary PIC/FLIP weight chosen to change the behavior of the fluid. This interpolated updated velocity is the new particle velocity.

Unlike the particle-to-grid transfer in Section 2.1, this step is easily parallelizable. Each particle can directly look up the closest grid elements, and therefore all writes to a particular particle can be performed easily by a single thread. I implement this step by dispatching one thread per particle to compute that particle's updated velocity.

2.7 Particle Advect

In the PIC scheme, advection simply means performing time integration on particle velocities. In my implementation, I use simple Explicit Euler integration. Ideally, I would use a more accurate scheme such as RK2 or RK4. However, I have found that other inaccuracies outweigh this issue; for now, Explicit Euler works well enough in practice.

3 Screen-Space Fluid

The particles used for advection in the PIC method already provide a convenient method with which we can visualize the fluid simulation. However, to make the simulation more applicable to real-time uses such as interactive media and video games, I wanted to improve the visual appearance of the fluid beyond a collection of points. I implement the Screen Space Fluid Rendering technique as presented by Simon Green for NVIDIA at GDC 2010 [4]. Below, you can see the same fluid configuration visualized as particles, and then with Screen Space Fluid Rendering (SSF).

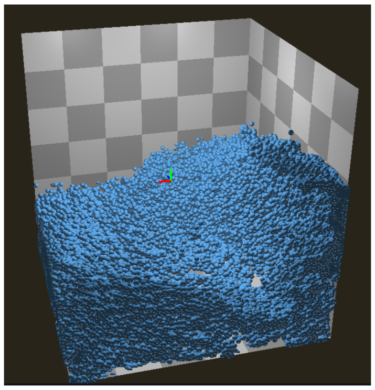

Figure 3: Fluid particles rendered as impostor spheres.

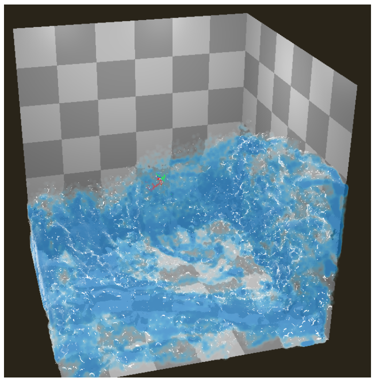

Figure 4: Fluid particles rendered using Screen Space Fluid Rendering.

Notice how this technique not only improves the visual appearance of the fluid particles, but also provides a visualization of the fluid density and overall volume of the fluid.

To provide background on this technique, I recommend reading Green's original presentation [4]. Below, I will briefly explain the rendering steps in my implementation:

1. Render background scene into scene buffer
2. Additively blend sphere volumes into "thickness" buffer
3. Write per-fragment sphere eye-space positions and clip-space depths into position and depth buffer A
4. Apply bilateral filter (surface blur) to sphere eye-space position depths, write updated position data to position buffer B
5. Render screen-space fluid to screen; for each fragment:
   (a) Sample thickness from thickness buffer
   (b) Sample eye-space position from position buffer B
   (c) Compute eye-space normals using finite differences of eye-space positions
   (d) Transform normals into world space using view matrix
   (e) Compute "fake" refraction by sampling background color from scene buffer at an offset based on the fluid normal
   (f) Compute transmitted color using refracted color and Beer's law to compute light absorption
   (g) Compute fresnel term and blend transmitted, reflected color

4 Conclusion

My implementation achieves interactive performance on commodity GPU hardware. Using my NVIDIA GTX 1070, I can simulate and render a 24x24 fluid grid with over 100,000 fluid particles at 60 frames per second. There are still some bugs remaining in the simulation as a result of time constraints. Additionally, I have left out some interesting features, such as a better time integrator for particle advection. The simulation would also benefit from time substeps and particle resampling, both of which would reduce issues with the fluid compressing slowly over time as a result of particles becoming more densely-packed. Overall, my implementation successfully demonstrates a proof-of-concept real-time fluid simulation that could be used in games and interactive media.

References

[1] J. U. Brackbill and H. M. Ruppel. 1987. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics* 65, 2 (Aug. 1987).

[2] Robert Bridson. 2016. *Fluid Simulation for Computer Graphics* (2nd. ed.). CRC Press, Boca Raton, FL.

[3] Jeff Dombroski, Taylor Gregory, Monique Shotande, and Eric Rackear. 2013. Marker-and-Cell Method (MAC). (October 2013). Retrieved from http://plaza.ufl.edu/ebrackear/

[4] Simon Green. 2010. Screen Space Fluid Rendering for Games. Presentation. At *Game Developers Conference 2010*. Retrieved from http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf

[5] Francis H. Harlow and J. Eddie Welch. 1965. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *The Physics of Fluids* 8, 12 (1965), 2182-2189.