Computational Fluid Dynamics With a Paper Airplane

Tasada, Daniel Tse, Nathan

January 30, 2025

Abstract

In this paper, we investigate the relationship between an airplane's shape and its performance. Our results show that …

Contents

1 Introduction

This thesis covers the simulation of the aerodynamics of an airplane, using our own Computational Fluid Dynamics model (CFD).

The goal is to simulate the airflow around an airplane's body. CFD is a branch of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. It is used in many fields, including aerospace engineering, automotive engineering, and meteorology.

The application of CFD to an airplane is important because it allows testing of a model's aerodynamic performance without building a physical model, or have to set up a wind tunnel. The practical alternative is much more expensive and time-consuming.

The final aim is to determine how the shape of an airplane's body affects its performance. Part of the project is to dynamically generate 3D models of airplanes, and simulate the airflow around them. The intention is to use machine learning to optimize the shape of the airplane's body to maximize performance.

CFD is challenging in the sense that it requires a good understanding of fluid dynamics, as well as a good understanding of the math involved. CFD is also very expensive from a computational perspective, so code optimization is important.

The thesis questions are the following:

- How do the different aspects of fluid dynamics work and how do we implement it in a computer program?
- How do we dynamically generate 3D models?
- How does an airplane's wing shape influence its performance?

2 Execution

2.1 Preliminary: Lagrangian Fluid Simulation

One method of simulating fluid dynamics is the Lagrangian method. This method models the fluid as a particle collision system, where the air is represented by particles that interact with each other to emulate a fluid. Our implementation is based on Rigid Body Collision Resolution (Hakenberg, 2005). We used this paper as a guide for all the math involved.

The math relies on the momentum, inertia, and velocity of the particles to calculate the collision normal and point of contact. The collision normal is the direction in which the particles are moving away from each other, and the point of contact is the point at which the particles collide.

The following variables are necessary to perform the calculations:

Angular momentum L $(kg \cdot m^2/s)$: $L = mvr$;
Inertia tensor I $(kg \cdot m^2)$: $I = \frac{L}{\omega}$;
Angular velocity ω (rad/s) : $\omega = \frac{\Delta\theta}{\Delta t}$;

Collision normal($n \in \mathbb{R}^3$) in world coordinates away from body;
Point of contact ($r_i \in \mathbb{R}^3$) in world coordinates with respect to p_i;
Orientation ($R_i \in SO(3)$) transforming from object to world coordinates;

Where i represents one of two particles in a given collision:

Velocity after collision	$\tilde{v}_i$,
Angular velocity after collision	$\tilde{\omega}_i$,
Constant	λ,

The following formulas represent the relation between particles:

$$\begin{gathered}
\tilde{v}_1 = v_1 - \frac{\lambda}{m_1} n; \\
\tilde{v}_2 = v_2 + \frac{\lambda}{m_2} n; \\
\tilde{\omega}_1 = \omega_1 - \Delta q_1; \\
\tilde{\omega}_2 = \omega_2 + \Delta q_2; \\
\text{where } q_i := I_i^{-1} \cdot R_i^{-1} \cdot (r_i \times n), \\
\text{and } \lambda = 2 \frac{n v_1 - n v_2 + \omega_1 I_1 q_1 - \omega_2 I_2 q_2}{(\frac{1}{m_1} + \frac{1}{m_2}) n^2 + q_1 I_1 q_1 + q_2 I_2}
\end{gathered}$$

This was implemented using Go and raylib, and the code is available at github.com/dtasada/paper at the **`lagrangian-go`** branch.

When dealing with a lot of particles, the simulation stops performing as well, because the number of calculations and iterations of the formulas listed above has a time complexity of $O(n^2)$, where n is the number of particles. This is because every particle handles collisions with every other particlee every single frame. This can be elegantly mitigated by implementing a three-dimensional grid system, where every particle is assigned to a cell in the grid. This way, particles only need to check for collisions with other particles in the same cell or neighboring cells. If the particles are evenly distributed within the container, each particle only needs to handle collisions with particles in its own cell, and its 26 neighboring cells. This reduces the time complexity to $O(n)$.

The simulation was unstable at higher densities, where particles would phase into each other instead of cleanly bouncing. The bug is likely quite simple to fix, but we got distracted by another method. Despite the optimizations we made, Lagrangian simulation is still quite computationally expensive, and Go probably wasn't doing it any favors (Go isn't very memory efficient). In the end it was put aside, and the Lagrangian implementation was never completed.

2.2 Eulerian Fluid Simulation

The other method is the Eulerian method. This method models the fluid as a cellular automata. The fluid is represented by a grid of cells, each of which has its own properties and interacts with its neighbor cells, creating a Cartesian mesh-style field.

Our fluid simulation involves a three-dimensional grid of cells, each of which have velocity and density fields. Each frame, the cells interact with each other according to the Navier-Stokes equations.

The specific math involved in the base equations for the fluid simulation is quite complex, and we couldn't have derived it on our own, so we used Mike Ash's implementation Fluid Simulation for Dummies (Ash, 2006) as the bedrock of our program. Mike Ash's code is in turn based on Jos Stam's brilliant work on Real-Time Fluid Dynamics for Games (Stam, 2003).

2.2.1 The Math

In Eulerian fluid dynamics we represent fluids with a velocity vector field. This means we assign a velocity vector to every point in space. The Navier-Stokes equations show us how these velocity vectors evolve over time with an infinitely small timestep.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u + v\nabla^2 + f$$

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla)\rho + k\nabla^2\rho + S$$

The first equation shows the change in velocity in a compact vector notation. Unlike in Lagrangian fluid simulation, in Eularian fluid simulation the fluid is not represented by individual particles. Thats why, instead, fluid density is used, which tells us the amount of particles present in a point in space. The second equation represents the change of this density. The reader is not expected to fully understand these equations because they are, as mentioned earlier, very difficult to understand. But it should be noted that the two equations above look a lot like eachother, as this was helpful in developing the simulation. (Stam, 2003)

2.2.2 The Implementation

Introduction to the code

Our project is structured as follows:

```
paper
├── neural ..................................... Contains ML engine
└── simulation/ ................................ Contains the physics simulation code
    ├── include/ ............................... Our own engine headers
    ├── resources/ ............................. Resources like images, fonts, shaders
    ├── src/ ................................... Actual engine source
    ├── config.toml ............................ Configuration file
    └── Makefile
```

This codebase is written in C++ and uses raylib again because it's a really simple and powerful library that allows us to focus on the simulation itself, as it provides a bunch of tools to easily render 3D graphics. The reason this codebase is in C++ was initially because we never got the Eulerian model to work in Go, likely due to a mistake of our

own. Nevertheless we think C++ was the right choice, as it's more appropriate for the vector physics and low overhead we need for the simulation.

The main fluid simulation object we use for the simulation is the **`Fluid`** class. The implementation is is available at **`/simulation/src/Fluid.cpp`**. The interface is as follows:

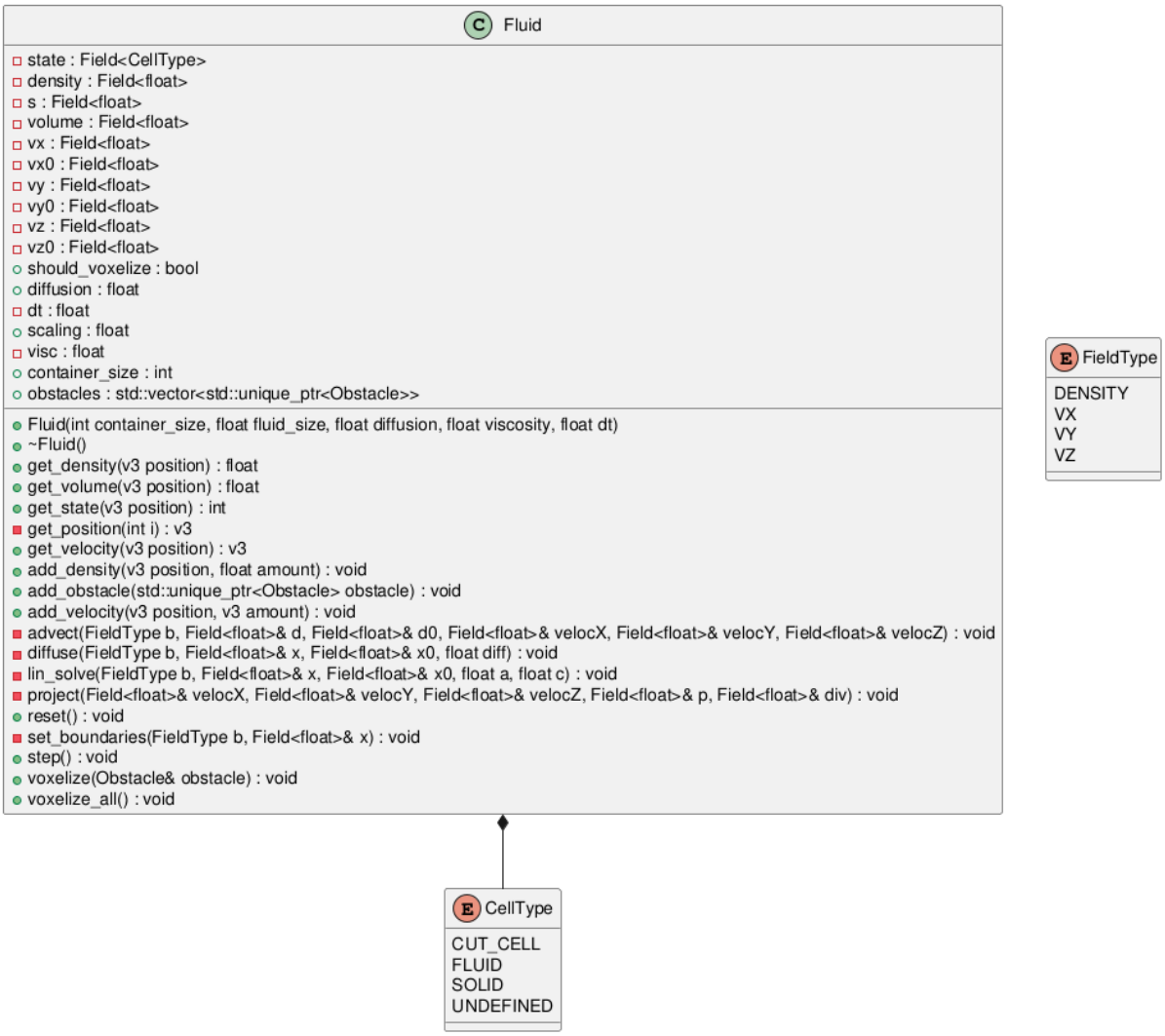

Figure 1: Fluid class diagram

This class contains the vector fields for the container as well as all the functions related to the physics simulation, including the advection, diffusion and projection procedures. These functions are the core of the CFD and this is the part that we borrowed from Mike Ash.

Figure 2: Basic fluid simulation

Voxelizing Geometry

The next step is to implement geometry collision. The graphics library we're using, Raylib, has a function to load geometry from object files. Loading in a torus is easy enough, but the real challenge lies in communicating to the CFD that there's a torus in a given position and it behaving correspondingly.

First, the 3D object that Raylib loads has to be converted to a BVH (Bounding Volume Hierarchy) object. This is a tree structure that contains the object's geometry in a way that makes it easy to check if a point is inside the object. This format is what FCL (Flexible Collision Library) uses to check for collisions. We are using FCL to easily process flexible collisions between 3D objects.

The next step is to write a voxelization function. Voxelization takes a 3D object's mesh data and checks the previously established 3D vector fields to see which if the object is in a given cell. FCL performs collision detection between two 3D objects. In this case those two objects are our obstacle (like a torus or a plane), and whichever cell is being iterated over. If the object is in a cell, we can pass a condition to the CFD. This is done for every cell in the grid, and the result is a 3D boolean array that represents the object. The source code is at `Fluid::voxelize`. Each obstacle is passed to this function, and as a result we get a 3D boolean field saved at `Fluid::state`. Below is a figure of the voxel map, with cells classified as solid highlighted in blue.

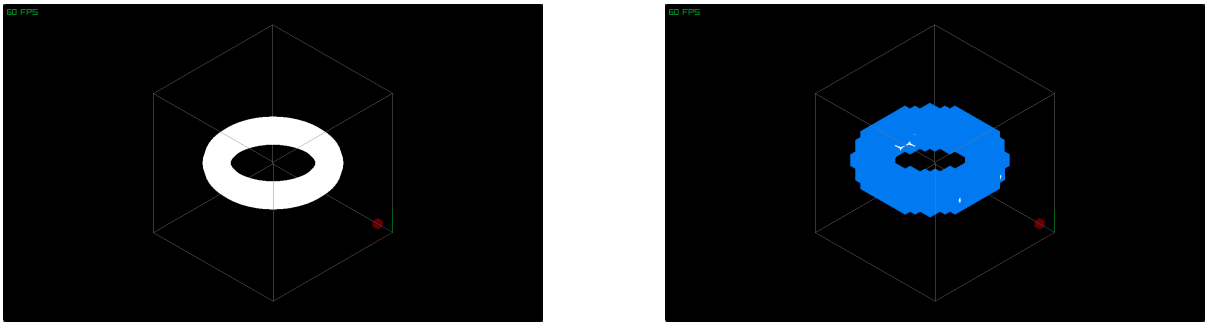

Figure 3: Voxel map

Implementing obstacle collision

Each cell of the grid is now one of three states: `FLUID`, `SOLID`, or `CUT_CELL`. This method of classifying cells into the three states is called the Cut-Cell method.

1. Fluid cells
    - Cells that are fully inside a fluid region

- Normal Navier-Stokes equations apply

2. Solid cells
   - Cells thar are fully inside an obstacle
   - Applies no-slip condition: velocity is zero
3. Cut-cells
   - Cells partially inside an object
   - Modify fluid calculations

3 Results

Nam quis enim. Quisque ornare dui a tortor. Fusce consequat lacus pellentesque metus. Duis euismod. Duis non quam. Maecenas vitae dolor in ipsum auctor vehicula. Vivamus nec nibh eget wisi varius pulvinar. Cras a lacus. Etiam et massa. Donec in nisl sit amet dui imperdiet vestibulum. Duis porttitor nibh id eros.

3.1 Conclusion

Nam quis enim. Quisque ornare dui a tortor. Fusce consequat lacus pellentesque metus. Duis euismod. Duis non quam. Maecenas vitae dolor in ipsum auctor vehicula. Vivamus nec nibh eget wisi varius pulvinar. Cras a lacus. Etiam et massa. Donec in nisl sit amet dui imperdiet vestibulum. Duis porttitor nibh id eros.

References

Ash, M. (2006). Fluid simulation for dummies. www.mikeash.com/pyblog/fluid-simulation-for-dummies.html

Gonkee. (2021). *Coding a fluid simulation with my last 2 brain cells*. https://www.youtube.com/watch?v=uG2mPez44eY

Gradience. (2024). *Teaching myself c so i can build a particle simulation*. www.youtu.be/NorXFOobehY

Hakenberg. (2005). Rigid body collision resolution. www.hakenberg.de/diffgeo/collision/rigid_body_collision_resolution.pdf

Ingram, D. M., & Causon, D. (2003). A cartesian cut cell method for incompressible viscous flow. www.researchgate.net/publication/222518000

Lague, S. (2023). *Coding adventure: Simulating fluids*. www.youtu.be/rSKMYc1CQHE

Library, F. C. (n.d.). *Flexible collision library*. https://github.com/flexible-collision-library/fcl

Müller, M. (2022). *17 - how to write an eulerian fluid simulator with 200 lines of code*. https://www.youtube.com/watch?v=iKAVRgIrUOU

Shiffman, D. (2019). *Coding challenge #132 fluid simulation*. www.youtu.be/alhpH6ECFvQ

Stam, J. (2003). Real-time fluid dynamics for games. www.dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf

Tucker, P. (2000). A cartesian cut cell method for incompressible viscous flow. www.sciencedirect.com/science/article/pii/S0307904X00000056

Unknown. (2012). Lattice-botlzmann fluid dynamics. www.physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf

Wikipedia. (2024). Navier-stokes equations. https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations

Work, P. (2023). *Writing a physics engine from scratch - collision detection optimization.* www.youtu.be/9IULfQH7E90