Computational Fluid Dynamics With a Paper Airplane

Tasada, Daniel Tse, Nathan

December 13, 2024

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 Computation Fluid Dynamics (or CFD) model.

The goal is to calculate the ideal shape of an airplane for its aerodynamic performance. We aim to do this by creating a Machine Learning model, and feed it the results of the CFD, which should result in our final model.

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

Our first attempt at a fluid simulator was using a technique called Lagrangian fluid simulation. This involves simulating the fluid as a particle collision simulation. It regards fluid dynamics, as particle physics, where every air molecule is a ”particle”. We start with a container, and a bunch of air molecules, all of which interact with each other to create a fluid. This interaction is effectively the collisions between particles. It turns out that simulating particle dynamics in two dimensions is pretty straightforward. But when you throw in that Z-axis, it gets a lot harder. The main article we used for this is Rigid Body Collision Resolution (Hakenberg, 2005).

Here is a set of formulas necessary to perform the calculations that describe the behavior of a pair particle before and after their collision:

The following variables are necessary to perform the calculations:

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

Collision normal($n \in \mathbb{R}^3$) in world coordinates away from body 1;
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:

$$\begin{gathered}
\tilde{v}_1 = v_1 - \frac{\lambda}{m_1} n; \\
\tilde{v}_2 = v_2 + \frac{\lambda}{m_2} n; \\
\tilde{\omega}_1 = \omega_1 - \Delta q_1; \\
\tilde{\omega}_2 = \omega_2 + \Delta q_2; \\
\text{where } q_i := I_i^{-1} \cdot R_i^{-1} \cdot (r_i \times n), \\
\text{and } \lambda = 2\frac{nv_1 - nv_2 + \omega_1 I_1 q_1 - \omega_2 I_2 q_2}{(\frac{1}{m_1} + \frac{1}{m_2})n^2 + q_1 I_1 q_1 + q_2 I_2}
\end{gathered}$$

After lots of trial and error, we were able to successfully implement the math. We did this using Go and raylib, and the code is available at `github.com/dtasada/paper` at the `lagrangian-go` branch. But we encountered a simple issue where the particles would phase into each other, rendering the particle simulation worthless. This could be because of a too small time step, or a logic error in the collision detection. This should be fixable by adding more simulation steps, which means that each particles solves its collisions more than once per collision, but we weren't able to successfully implement this. In the end, we ended up scrapping the Lagrangian simulation model.

2.2 Eulerian Fluid Simulation

Next we tried a method that we'd had our eyes on for a while. This was had originally been our first choice, but we switched to Lagrangian due to problems I'll explain shortly. It's called Eulerian fluid simulation, and the way this model works is that it sees fluid as a grid of cells that react to each other. Eulerian fluid simulation is an application of cellular automata, which is a computation model used in physics, biology, and many other applications.

Our fluid simulation involves a two or 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 math involved in the Navier-Stokes equations is very complex and difficult to understand without a good background in physics and differential equations. That is why we have humbly borrowed most of the math from Mike Ash's Fluid Simulation for Dummies (Ash, 2006), which is in turn based on Jos Stam's brilliant work on Real-Time Fluid Dynamics for Games (Stam, 2012).

2.2.1 The Math

In Eularian 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, 2012)

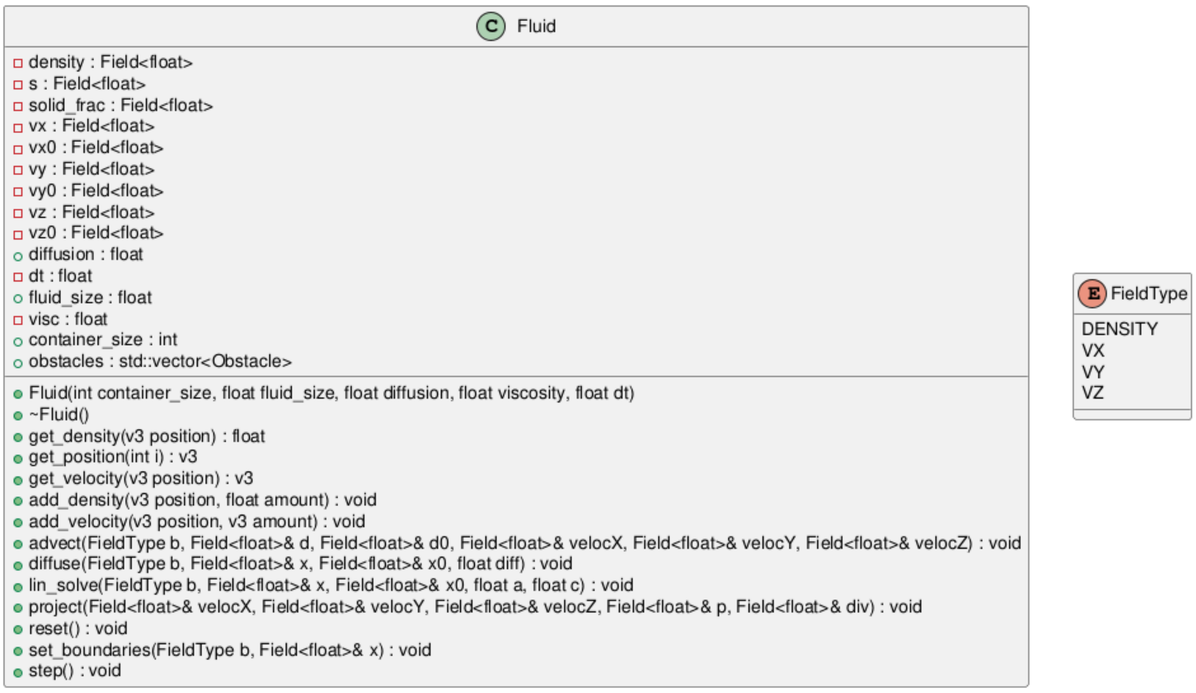
2.2.2 The Implementation

Our code is structured as follows:

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

The code is written in C++, and uses the raylib library for easy 3D rendering. We originally wanted to use Go because of preference and developer ergonomics, which is what we did with the first Lagrangian model, but we ended up writing the physics simulation in C++ because of the way that Jos Stam's implementation utilizes 3D arrays, which we found not to be viable in Go, which is why we switched to the Lagrangian model in the first place.

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


C Fluid
density : Field<float>
s : Field<float>
solid_frac : Field<float>
vx : Field<float>
vx0 : Field<float>
vy : Field<float>
vy0 : Field<float>
vz : Field<float>
vz0 : Field<float>
diffusion : float
dt : float
fluid_size : float
visc : float
container_size : int
obstacles : std::vector<Obstacle>
Fluid(int container_size, float fluid_size, float diffusion, float viscosity, float dt)
~Fluid()
get_density(v3 position) : float
get_position(int i) : v3
get_velocity(v3 position) : v3
add_density(v3 position, float amount) : void
add_velocity(v3 position, v3 amount) : void
advect(FieldType b, Field<float>& d, Field<float>& d0, Field<float>& velocX, Field<float>& velocY, Field<float>& velocZ) : void
diffuse(FieldType b, Field<float>& x, Field<float>& x0, float diff) : void
lin_solve(FieldType b, Field<float>& x, Field<float>& x0, float a, float c) : void
project(Field<float>& velocX, Field<float>& velocY, Field<float>& velocZ, Field<float>& p, Field<float>& div) : void
reset() : void
set_boundaries(FieldType b, Field<float>& x) : void
step() : void
E FieldType
DENSITY
VX
VY
VZ

The class contains the cell property fields as well as all the functions related to the physics simulation, including the advection, diffusion and projection procedures. With this we have a basic working fluid simulator.

The next step is to add geometry. We did this by adding a boolean field to the Fluid properties and some getters and setters.

```
class Fluid {
  private:
   // rest of private members
   bool *solid;

  public:
   // rest of public members
   bool is_solid(v3 position);
   void set_solid(v3 position, bool set);
}
```

Then we change the advection and boundary handling functions to account for solid boundaries. This is done by checking whether each cell is solid, and if so, the cell properties aren't advected.

```
// in advection function, for each cell:
if (solid[IX(i, j, k)]) {
   if (b ≠ FieldType::DENSITY) { // For velocity components
```

```
        d[IX(i, j, k)] = 0;
    }
    continue;
}

// in boundary function, for each cell:
if (solid[IX(x, y, z)]) {
    // For velocity components, enforce no-slip condition
    if (b == FieldType::VX) f[IX(x, y, z)] = 0; // x velocity
    if (b == FieldType::VY) f[IX(x, y, z)] = 0; // y velocity
    if (b == FieldType::VZ) f[IX(x, y, z)] = 0; // z velocity

    // For density and pressure, use average of neighboring
    // non-solid cells
    if (b == FieldType::DENSITY) {
        float sum = 0;
        int count = 0;

        if (!solid[IX(x-1, y, z)]) { sum += f[IX(x-1, y, z)]; count++; }
        if (!solid[IX(x+1, y, z)]) { sum += f[IX(x+1, y, z)]; count++; }
        if (!solid[IX(x, y-1, z)]) { sum += f[IX(x, y-1, z)]; count++; }
        if (!solid[IX(x, y+1, z)]) { sum += f[IX(x, y+1, z)]; count++; }
        if (!solid[IX(x, y, z-1)]) { sum += f[IX(x, y, z-1)]; count++; }
        if (!solid[IX(x, y, z+1)]) { sum += f[IX(x, y, z+1)]; count++; }

        f[IX(x, y, z)] = count > 0 ? sum / count : f[IX(x, y, z)];
    }
}
```

This allows us to add a cube of a given size at a given position, which the fluid will treat as a solid object. Unfortunately, a plane is more complex than a cube. There are a few ways to handle this. The most obvious and least efficient is to raise the resolution of the grid. Currently we've been experimenting with a 24x24x24 or a 32x32x32 grid, and that's already not as performant as we'd like. A 24x24x24 grid has to iterate $24^3 = 13824$ times per frame. And to simulate complex shapes, the higher resolution the better. But simply increasing the resolution to something like 128x128x128 or higher just isn't viable.

We're still implementing this at the time of writing the concept, but the goal is to use the method explained above in combination with the Cut-Cell method, which will dynamically create more of these cells along the borders of the model, in this case a plane. This would create a high-resolution grid near the important and complex areas of the simulation, while preserving resources by keeping the unimportant areas (like large patches of just air) from being computed.

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

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

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. (2012). Real-time fluid dynamics for games. www.dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf

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