A High Quality, Eulerian 3D Fluid Solver in C++

A Senior Project

presented to

the Faculty of the Computer Science Department of

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science, Software Engineering

by

LeJon McGowan

November, 2017

Table of Contents

Introduction

Fluids are a part of everyday life, yet are one of the hardest elements to properly render in computer graphics. Water is the most obvious entity when thinking of what a fluid simulation can achieve (and it is indeed the focus of this project), but many other aspects of nature, like fog, clouds, and particle effects. Real-time graphics like video games employ many heuristics to approximate these effects, but large-scale renderers aim to simulate these effects as closely as possible.

In this project, I wish to achieve effects of the latter nature. I seek to further my knowledge on a particular type of fluid simulation, by interpreting each attribute of a fluid as a 3-dimensional grid that you change and query as the simulation evolves.

Requirements

I aim to implement a 3-dimensional fluid solver in C++ that can be used to generate model realistic flow in a scene. While I am not necessarily aiming for anything close to real-time with my solution, I'd want it to render a medium-size scene in a somewhat reasonable amount of time, and will keep that in mind it my implementation. The solver will be given some basic parameters detailing the resolution, duration, and initialization of the scene, and from there will iterate through the scene based on a given delta time (which in my case was a time step meant to match 60Hz monitors). At the end of each delta, measures will be take to translate the results at this particular time into a 3-dimensional mesh format that can be processed by external programs for other purposes, like translating frames into an animation. Though I have no problem with relying on external dependencies for mathematical calculations, I desire to implement the core of the solver from the base language, including any (non-mathematical) data structures needed to store the fluid's state. This core solver will be developed with external processes, like mesh generation, in mind.

Architecture

Math

The math used in the application is relatively straightforward. All the math needed outside of the C++'s built in functions are vectors and matrices. Given the integral, but ultimately small part of the application, I opted to use one of the many available libraries. Usually, I would make use of the math library GLM, a header-only library with a fairly elegant notation modeled after the vectors and matrices used in the OpenGL Shading language (GLSL). However, GLM, being modeled as it is, is made to work with vectors and matrices of up to 4 dimensions. I anticipated that I may have to make use of large scale systems of equations for some of the solvers, so I opted for a more powerful library. As such, I decided to make use of Eigen. Eigen, while having more verbose notation (there is no sense of x/y/z/w in Eigen, vectors in Eigen are purely interpreted as enhanced arrays), is much more versatile and has a dozen different, highly optimized solvers, including some made to work with sparse matrices (another feature that GLM lacks).

Grids

Being an Eulerian implementation, the most important data structure to use is a grid structure. There is a base, stateless, implementation of a grid that allows for querying of common attributes, including the grid's resolution, spacing size, and origin.

From there, there are two different types of grids that inherit from this base class: Scalar and Vector. These are, expectedly, used to give a discrete representation of a scalar and vector field of the fluid's various attributes. These two classes are where the state of the data is held. The data is held in a flat, 1-dimensional array (real values for scalar, and 3d vectors for vector), but has utility methods to access this data as if it was in 3-D space. From here, each type of grid has two subclasses implemented that are based on where to store the discrete data: in the center of each grid block, or on the vertices.

One special grid in particular that is used for storing velocity data is called a Marker and Cell (MAC) grid. In this kind of grid, scalar values are at the center of 3 of the faces, which are used to stably sample the middle of the cube for values. Any operations that need to apply to these kinds of grids typically operate on each of these 3 separate markers in succession.

The most important aspect of these grids is the ability to sample for values that are not on the immediate discrete points. These values are approximated using interpolation. There are two different sampling built in, depending on the amount of accuracy needed: a simple implementation of trilinear interpolation, and a more accurate cubic interpolation implemented using Catmull-Rom splines. For many methods (including approximating vector calculus functions), the former suffices as long as the grid spacing chosen is not too large, but other aspect of the solver like the advection needs more refined results and uses the latter.

The final system is made to use a variety of grids. Velocity is always a given, but users may want to add additional parameters, like a grid for buoyancy for smoke simulations. As such, there is one "manager" class that stores a map of the relevant grids to process. When it comes time to advect, the advection will iterate over all the grids in the manager and apply the flow to each grid, and finally advect the velocity.

Solvers

Each solver is stored in its own class, with a single public method used to calculate the relevant part of the fluid. Though it is possible to accomplish all of the fluid calculations in one pass, splitting the process into sequentially solving these processes piece-by-piece allows for a much more scalable system, once that can apply new and different techniques further down the line. For example, one may want to test a simple implementation of viscosity calculations using a forward Euler. However, one later finds a need for more stable calculations as simulations become longer. As such, I decided to make use of the backwards Euler method to achieve this, and was able to change the implementation of the Viscosity solver without altering the rest of the system. This structure allows for the modification of one method while leaving the rest of the system intact.

Mesh reconstruction

After the solving is done, the final task to perform is converting the graph representing the results into some sort of surface for that frame. In 2D, this is a relatively simple matter, as one only needs to interpret a 2D graph like pixels on a screen and color pixels based on where the fluids are in that moment.

However, this is an order of magnitude more difficult in 3D. The most common way for a computer to represent a 3D mesh is through the use of a series of triangles, but getting to even this point in a non-trivial matter. This is slightly outside of the scope of this solver, but at the same time a very important aspect. As such, I made use of an existing solution that implemented the Marching Cube method, a method that walks through a series of scalar values (in this case, values representing a signed distance field) and constructs triangles based on where it determines the surface to be. This is a small module of the Jet framework. With a small bridge method, I made use of this part of the framework to construct an output an .obj file for later processing in Blender modeling application.

Main Loop

With all the systems setup, the basic procedure of running the solver can work. The algorithm of how to run the simulations is as follows:

1. Initialize the scene and relevant parameters for the solver
2. For each frame of the solver
    a. Apply external forces (usually gravity, but it can be a variety of methods) to the solver
    b. Apply viscosity to the system, if needed
    c. Apply pressure to the system
    d. Based on the final velocity from these calculations use this final flow to move other grids (and finally, the velocity itself) in the proper way. This concept is known as advection.
    e. In the case of our signed distance field mesh, correct the sdf to preserve its correctness.
3. Finally, render each frame into a mesh file for external processing (in this case, for Blender to render).

Implementation

External Forces

The first part of the solver is applying external forces to the current velocity. The most common force to model is from the acceleration of the fluid due to gravity. This force is modeled by a simple equation of:

$$F = mg$$

Where m is the mass of the fluid, and g is the acceleration due to gravity, which on Earth is approximately 9.81 m/s. Calculating the velocity over a given timeframe with this model is trivial. Since the acceleration is a known constant, we apply the kinematic equation of:

$$v = at$$

By simply substituting a with g and adding the resulting velocity to the current velocity in that frame.

Viscosity

Viscosity can be thought of as the internal "friction" within the fluid. This property is what makes thicker liquids like honey flow slower. One can consider this friction as the "blurring" of neighboring velocities, so the final equation ends up as:

$$a_{\mathrm{v}} = \mu \nabla^2 u$$

Where ν (nu) represents viscosity, and u represents velocity (with “u” being used instead of “v” as to not confuse it with nu in this case). μ is the viscosity constant that varies from fluid to fluid. For example, μ for water is 0.00089 Pa-s and ranges from 2-10 Pa-s for honey. If we apply the same kind of step we did for gravity, we end up with the final velocity of

$$u_\nu = \mu\nabla^2 u\Delta t$$

However, complications arise overtime from just adding such this velocity like we did for gravity. Unless we keep our timestep extremely small (and as a result, increase the computation time for a longer scene), the values given in this result will start to accumulate error until the scene grows too unstable. The current, forward method of adding new values to the old is inadequate here. Instead, we should prompt for a backwards method of solving this final velocity while retaining stability, which results in this model:

$$u^{n+1} = u^n + \mu\nabla^2\Delta t u^{n+1}$$

Pressure

The most important part of the solver is to calculate the amount of pressure between a fluid based on its neighbors and accelerate the fluid accordingly. By definition, pressure is based on the ratio between the force exerted over a given area:

$$p = \frac{F_p}{A_p}$$

The gradient of this pressure creates gradient force exerted on the fluid. With some algebraic manipulation of this resulting gradient force:

$$F_p = ma_p = \nabla p A_p$$

we can derive the acceleration of the resulting gradient force:

$$a_p = \frac{\nabla p A_p}{m} = \frac{\nabla p l^2}{\rho l^3} = \frac{\nabla p}{\rho l}$$

Where ρ is the density of the fluid (derived from the fact that mass is proportional to the density and volume of a substance), and l is the distance between two portions of a substance. As this distance reaches a limit of 0, the final acceleration converges to:

$$a_p = \frac{\nabla p}{\rho}$$

Finally, pressure acts opposite to the external force (e.g. gravity) acted upon it. So, the final equation is negated:

$$a_p =- \frac{\nabla p}{\rho}$$

Much like viscosity, calculating this gradient in a discrete sense is not trivial. And a forward Euler approach will diverge outwards in the long term. In a similar fashion to viscosity, we can derive a backwards Euler approach for this, which results in:

$$\nabla^2(\frac{\Delta t}{\rho}p) = \nabla \bullet V^n$$

Where V^n is the velocity of the current frame. This final equation, called the pressure Poisson equation, creates a linear system to find the one unknown, p. Given that this linear system is applied to all grid points in the 3-d grid, the resulting matrix built is extremely large (and like the viscosity, filled with many zeros, since at most only the element and its neighbors are taken into account on each row), optimizations should be taken to solve this in a timely matter.

Advection

With all the above steps performed, the final step is to take the advect the actual fluid to a new location based on the new, calculated velocity. The semi-lagrangian method is used to backtrace through the flow in order to find the appropriate place to sample. Once found, this sample is assigned to the respective discrete values in the grid. In this case, the upwind method is used to carry out this backtrace.

Future considerations

Parallelization/Hardware Acceleration

One of the biggest advantages of grid based solutions is the potential for parallelization. Many of the operations used in each step of the solver involves taking an input of the current grid, iterating over each discrete point (which, for a 3-dimensional grid is of O^3 complexity), and returning the output of the grid after operations are performed. Since many of these operations rely on only the unaltered input, this makes the resulting calculations embarrassingly parallel. As such, significant speed-ups can be gained if implementations were either run with some threaded solution (from C++11's own threads to a more complex schedule like Intel Threading Blocks) or through the GPU using the respective API, like Nvidia CUDA or OpenCL.

Configuration options

Currently, values are hard-coded into the main file in order to configure the scene. As such, compilation of the program has to occur every time the scene or some parameter is changed, Steps can be taken to have these setup at runtime, be it through command-line arguments, an input file that describes the scene, or perhaps use of a simple scripting language that is interpreted at runtime to create the scene.

Custom mesh generation

One of the biggest external dependencies of the project is in the framework used to generate the mesh from the final 3d-grid. Unlike the other mathematical frameworks I used, this is a very specific implementation, and should ideally be replaced by my own implementation one day in order to more easily distribute the program to users.

Solid Boundaries

Moving boundaries were taken into consideration when constructing each component of the solver, but ultimately the solver has only been tested with trivial box boundaries, based on the given dimensions of the scene. There would need to be a few more modifications made at points where the boundary conditions are applied in order to properly simulate this behavior.

Conclusion

Fluid simulation is a vast sub-field with computer graphics, and this is only one possible way to approach calculating fluids Most popular implementations combine this grid-based approach with a Lagrangian, particle-based approach in order to achieve the best aspect of both types. Overall, this project was humbling, to say the least. Though this was meant to be a mid-scale project (there are many simple implementations out there), the amount of mathematical and computational knowledge and work needed to make this project work much more significant than expected. Even then. there are several features of liquids that my simulation fails to replicate. This was a very lucrative experience for me, but it also shows how much more of this topic there is to learn.

References

Bridson, Robert. *Fluid simulation for computer graphics*. CRC Press, Taylor & Francis Group, CRC Press is an imprint of the Taylor & Francis Group, an informa Business, 2016.

Dombroski, Jeff, et al. “Marker-and-Cell Method (MAC).” *Plaza.ufl.edu*, University of Florida, 17 Oct. 2013, plaza.ufl.edu/ebrackear/.

Gourlay, Dr. Michael J. “Fluid Simulation for Video Games.” Intel® Software, Intel, 12 Sept. 2014, software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1.

Kim, Doyub. *Fluid engine development*. Taylor & Francis, a CRC Press,Taylor & Francis Group, 2017.