

Fluid Simulation for Video Games (Part 21)

By Dr. Michael J. Gourlay

Recapitulation

We want games to be fun and look pretty and plausible.

Fluid simulation can augment game mechanics and enhance the aesthetics and realism of video games. Video games demand high performance on a low budget but not great accuracy. By *low budget*, I mean both computational and human resources: the game has to run fast, and it can't take a lot of developer or artist time. There are many ways to simulate fluids. In this series, I explain methods well suited to video games: cheap, pretty, and easily written.

If you want to simulate fluids in video games, you have to overcome many challenges. Fluid mechanics is a complicated topic soaked in mathematics that can take a lot of time to understand. It's also numerically challenging—naïve implementations are unstable or just behave wrong—and because fluids span every point in space, simulating them costs a lot of computational resources, both processing and memory. (Although fluid simulations are well suited to running on a graphics processing unit [GPU], in video games, the GPU tends to be busy with rendering.) The easiest and most obvious way to improve numerical stability—use more and smaller time steps—adds drastically more computational cost, so other techniques tend to be employed that effectively increase the viscosity of the fluid, which means that in-game fluids tend to look thick and goopy. The most popular techniques, like smoothed particle hydrodynamics, aren't well suited to delicate, wispy motions like smoke and flame. A simulation approach that could meet these challenges would help add more varieties of fluids, including flames and smoke, to video games.

To meet these challenges, I presented a fluid simulation technique suited to simulating fine, wispy motion and that builds on a particle system paradigm found in any complete three-dimensional (3D) game engine. It can use as many CPU threads as are available—more threads will permit more sophisticated effects.

This approach has many synergies with game development, including the following:

- Game engines support particle systems and physics simulations.
- CPUs often have unused cores.
- Fluid simulation readily parallelizes.

The approach I present uses the vortex particle method (VPM), an unusual technique that yields the following benefits:

- It's a particle-based method that can reuse an existing particle engine.
- Vortex simulations are well suited to delicate, wispy motion like smoke and flame.

- The simulation algorithm is numerically stable, even without viscosity, either explicit or implicit.

This series explains the math and the numerics. I presented variations on the theme of particle-based fluid simulation and included a model of thermal convection and combustion so that the system can also model flames. I showed two numerical approaches—integral and differential—compared their relative merits, and presented a hybrid approach that exploits the benefits of each approach while avoiding their pitfalls. The result is a fast (linear in the number of particles) and smooth fluid simulation capable of simulating wispy fluids like flames and smoke.

Despite its apparent success in certain scenarios, the VPM has significant limitations in other scenarios. For example, it has difficulty representing interfaces between liquids and gases, such as the surface of a pool of water. I took a brief detour into smoothed particle hydrodynamics (SPH) to explore one way you could join VPM and SPH in a common hybrid framework, but (to phrase it generously) I left a lot of room for improvement.

This series of articles leaves several avenues unexplored. I conclude this article with a list of ideas that I encourage you to explore and share with the community.

[Part 1](#) and [part 2](#) summarized fluid dynamics and simulation techniques. [Part 3](#) and [part 4](#) presented a vortex-particle fluid simulation with two-way fluid–body interactions that run in real time. [Part 5](#) profiled and optimized that simulation code. [Part 6](#) described a differential method for computing velocity from vorticity. [Part 7](#) showed how to integrate a fluid simulation into a typical particle system. [Part 8](#), [part 9](#), [part 10](#), and [part 11](#) explained how to simulate density, buoyancy, heat, and combustion in a vortex-based fluid simulation. [Part 12](#) explained how improper sampling caused unwanted jerky motion and described how to mitigate it. [Part 13](#) added convex polytopes and lift-like forces. [Part 14](#), [part 15](#), [part 16](#), [part 17](#), and [part 18](#) added containers, SPH, liquids, and fluid surfaces. [Part 19](#) provided details on how to use a treecode algorithm to integrate vorticity to compute vector potential.

Fluid Mechanics

With the practical experience of having built multiple fluid simulations behind us, I now synopsise the mathematical and physical principles behind those simulations. Fluid simulation entails running numerical algorithms to solve a system of equations simultaneously. Each equation governs a different aspect of the physical behaviors of a fluid.

Momentum

The *momentum equation* (one version of which is the famous Navier-Stokes equation) describes how momentum evolves and how mass moves. This is a nonlinear equation, and the nonlinearity is what makes fluids so challenging and interesting.

Vorticity is the curl of momentum. The *vorticity equation* describes where fluid swirls—where it moves in an “interesting” way. Because vorticity is a derivative of momentum, solving the vorticity equation is tantamount to solving the momentum equation. This series of articles exploits that connection and focuses numerical effort only where the fluid has vorticity, which is usually much sparser than where it has momentum. The vorticity equation also implicitly discards divergence in fluids—the part that relates to the compressibility of fluids. Correctly dealing with compressibility requires more computational resources or more delicate numerical machinations, but in the vast majority of scenarios pertinent to video games, you can neglect compressibility. So the vorticity equation also yields a way to circumvent the compressibility problem.

Advection describes how particles move. Both the momentum equation and the vorticity equation have an advective term. It’s the nonlinear term, so it’s responsible both for the most interesting aspects of fluid motion and the most challenging mathematical and numerical issues. Using a particle-based method lets us separate out the advective term and handle it by simply moving particles around according to a velocity field. This makes it possible—easy, in fact—to incorporate the VPM into a particle system. It also lets the particle system reuse the velocity field both for the “vortex particles” the fluid simulation uses and for propagating the “tracer” particles used for rendering visual effects.

The *buoyancy* term of the momentum and vorticity equations describes how gravity, pressure, and density induce torque. This effect underlies how hot fluids rise and so is crucial for simulating the rising motion of flames and smoke. Note that the VPM simulation technique in this series did not model pressure gradients explicitly but instead assumed that pressure gradients lie entirely along the gravity direction. This supposition let the fluid simulate buoyancy despite not having to model pressure as its own separate field. To model pressure gradients correctly, you must typically model compressibility, which, as mentioned elsewhere, usually costs a lot of computational resources. So by making the simplifying assumption that the pressure gradient always lies along the gravity direction, you see a drastic computational savings. Computing density gradients require knowing the relationship between adjacent particles. In this series, I presented two ways to solve this: a grid-based approach and a particle-based approach. The grid-based approach directly employs a spatial partitioning scheme that is also used in computing viscosity effects. The particle-based approach uses a subset of the algorithms that SPH uses. Both can yield satisfactory results, so the decision comes down to which approach costs less.

The *stress and strain* terms in the momentum and vorticity equations describe how pressure and shears induce motion within a fluid. This is where viscosity enters the simulation. Varying the magnitude and form of viscosity permits the simulation to model fluids ranging from delicate, wispy stuff like smoke or thick, goopy stuff like oil or mucous. The fluid simulation in this series used the particle strength exchange (PSE) technique to exchange momentum between nearby particles. This technique requires that the simulation keep track of which particles are near which others—effectively, knowing their nearest neighbors. I presented a simplistic approach that used a uniform grid spatial partition, but others could work, and this is one of the avenues I encourage you to explore further.

The *stretch and tilt* terms of the vorticity equation describe how vortices interact with each other over distance as a result of configuration. This is strictly a 3D effect, and it leads to turbulent motion. Without this effect, fluids would behave in a much less interesting way. The algorithms I presented compute stretch and tilt using finite differences, but others could work. At the end of this article, I mention an interesting side effect of this computation that you could use to model surface tension.

Conservation of Mass

The *continuity equation* states that the change in the mass of a volume equals inflow/outflow of mass through volume surfaces. As described earlier, the simulation technique in this series dodged solving that equation explicitly by imposing that the fluid is incompressible.

Equation of State

The *equation of state* describes how a fluid expands and contracts (and therefore changes density) as a result of heat. Coupled with the buoyancy term in the momentum equation, the equation of state permitted the algorithm to simulate the intuitive behavior that “hot air rises, and cold air sinks.”

Combustion

The *Arrhenius equation* describes how components of fluid transform: fuel to plasma to exhaust. It also describes how fluid heats up, which feeds into the equation of state to model how the fluid density changes with temperature, hence, causing hot air to rise.

Drag

Drag describes how fluids interact with solid objects. I presented an approach that builds on the PSE approach used to model viscosity: I treat fluid particles and solid objects in a similar paradigm. I extended the process to exchange heat, too, so that solid objects can heat or cool fluids and vice versa.

Spatial Discretization

Fluid equations operate on a continuous temporal and spatial domain, but simulating them on a computer requires discretizing the equations in both time and space. You can discretize space into regions, and those regions can either move (for example, with *particles*) or not (for example, with a fixed *grid*).

As its name suggests, the VPM is a particle-based method rather than a grid-based method. The algorithm I presented, however, also uses a uniform grid spatial partition to help answer queries about spatial relationships, such as knowing which particles are near which others or which

particles are near solid objects interacting with the fluid. Many spatial partitions are available and within each implementation are possible. For this article series, I chose something reasonably simple and reasonably fast, but I suspect that it could be improved dramatically, so I provide some ideas you can try at the end of this article.

Note: Other discretizations are possible—for example, in a spectral domain. I mention this in passing so that curious readers know about other possibilities, but for the sake of brevity I omit details.

Vortex Particle Method

In this series, I predominantly employed the VPM for modeling fluid motion, but even within that method, you have many choices for how you implement various aspects of the numerical solver. Ultimately, the computer needs to obtain velocity from vorticity, and there are two mathematical approaches to doing so: integral and differential. Each of those mathematical approaches can be solved through multiple numerical algorithms.

The integral techniques I presented are *direct summation* and *treecode*. Direct summation has asymptotic time complexity $O(N^2)$, which is the slowest of those presented but also the simplest to implement. Treecode has asymptotic time complexity $O(N \log N)$, which is between the slowest and fastest, and has substantially more code complexity than direct summation, but that complexity is worth the speed advantage. Besides those techniques, other options are possible that I did not cover. For example, *multipole methods* have asymptotically low computational complexity order but mathematically and numerically require much greater complexity.

The differential technique I presented entails solving a vector Poisson equation. Among the techniques I presented, this has the fastest asymptotic run time, and the math and code are not very complex. Based on that description, it seems like the obvious choice, but there is a catch that involves boundary conditions.

Solving any partial differential equation entails imposing boundary conditions: solving the equations at the spatial bounds of the domain. For integral techniques, the simplest conditions are “open,” which is tantamount to having an infinite domain without walls. The simulation algorithm handles solid objects, including walls and floors, which should suffice to impose boundary conditions appropriate to whatever scene geometry interacts with the fluid, so imposing additional boundary conditions would be redundant.

The Poisson solver I presented uses a rectangular box with a uniform grid. It’s relatively easy to impose “no-slip” or “no-through” boundary conditions on the box, but then the fluid would move as though it were inside a box. You could move the domain boundaries far from the interesting part of the fluid motion, but because the box has a uniform grid, most of the grid cells would have nothing interesting in them yet would cost both memory and compute cycles. So ideally you’d have a Poisson solver that supports open boundary conditions, which is tantamount to knowing the solution at the boundaries, but the Poisson solver is meant to obtain the solution and so is a cyclic dependency.

To solve this problem, I used the integral technique to compute a solution at the domain boundaries (a two-dimensional surface), and then used the Poisson solver to compute a solution throughout the domain interior. This hybrid approach runs in $O(N)$ time (faster than treecode) and looks better than the treecode solver results.

Assessment

The VPM works well for fire and smoke but does not work for liquid–gas boundaries. SPH works well for liquid–gas boundaries but looks viscous. My attempt to merge them didn’t work well, but I still suspect the approach has merit.

Further Possibilities

The techniques and code I presented in this series provide examples and a starting point for a fluid simulation for video games. To turn these examples into viable production code would require further refinements to both the simulation and the rendering code.

Simulation

Improvements to VPM

I implemented a simplistic uniform grid spatial partitioning scheme. A lot of time is spent performing queries on that data structure. You could optimize or replace it, for example, with a spatial hash. Also, you could switch the per-cell container to a much more lightweight container.

Although difficult, it’s possible to model those liquid–gas boundaries in the VPM. You could track surfaces by using level-sets, surface geometry to compute curvature, or curvature to compute surface tension and incorporate those effects into the vorticity equation. Computing curvature entails computing the Hessian, which is related to Jacobian, which is already used to compute strain and stress.

The VPM has a glaring mathematical problem: It starts with a bunch of small particles that carry vorticity in a very small region—so small it’s tempting to think of them as points. Vorticity mathematically resembles a magnetic field, and you could draw an analogy between these vortex particles and tiny magnets. These magnets, however, would have only a single “pole,” which is both mathematically and physically impossible. Likewise, there is no such thing as a vortex “point.” If you had only one vortex point, it would be possible for a vortex field to have divergence, and this is neither mathematically possible nor physically meaningful. And yet, this simulation technique has exactly this problem. One way to solve the problem is to use vortex filaments—for example, topologically forming loops. The vortex loops, being closed, would have no net divergence. (See, for example, “Simulation of Smoke Based on Vortex Filament Primitives” by Angelidis and Neyret.) The filaments could also terminate at fluid boundaries, such as at the interfaces with solid objects. The

most obvious example of that would be a rotating body: Effectively, the body has a vorticity and so vortex lines should pass through the body.

Note: The article “Filament-Based Smoke with Vortex Shedding and Variational Reconnection” as presented at SIGGRAPH 2010 got that wrong: The authors had rotating bodies within a fluid, but their vortex filaments did not pass through those bodies. They seem to have corrected that error in subsequent publications, and the YouTube* videos that had the error are no longer visible.

Other Techniques

Because SPH is also a fluid simulation technique that uses particles, my intuition is that it should complement the VPM so that some hybrid could work for both wispy, and liquid or goopy fluids. I would not call my attempt successful, but I hope it inspires future ideas to unify those approaches. Even though my implementation failed, I suspect that the basic idea could still be made to work.

This article series did not cover them, but grid-based methods work well in specialized cases, such as where potential flow is important, and for shallow-water waves. Similarly, spectral methods are capable of tremendous accuracy, but that is exactly what video games can forsake.

Rendering

In the code that accompanies these articles, most of the processing time goes toward rendering rather than simulation. That’s good news because the simplistic rendering in the sample code doesn’t exploit modern GPUs and so there’s plenty of opportunity to speed that up.

The sample code performs several per-vertex operations, such as computing camera-facing quadrilaterals. That code is embarrassingly parallel, so a programmable vertex shader could execute it on the GPU quickly because the GPU has hundreds or thousands of processing units.

It turns out, though, that adding more CPU cores to those routines that operate on each vertex doesn’t yield a linear speed-up, which suggests that memory bandwidth limits processing speed. Effectively, to speed up processing, the machine would need to access less memory. Again, a solution is readily available: Inside the vertex buffer, instead of storing an element per triangle vertex, store only a single element per particle. It could even be possible to transmit a copy of the particle buffer as is. Because you can control how the vertex shader accesses memory, that vertex buffer can be in any format you like, including the one the particle buffer has. This implies using less memory bandwidth.

Note that the GPU would likely still need a separate copy of the particle buffer, even if its contents were identical to the particle buffer the CPU used. The reason is that those processors run asynchronously, so if they shared a buffer, it would be possible for the CPU to modify a particle buffer in the middle of the GPU accessing that data, which could result in inconsistent rendering artifacts. In that case, it might be prudent to duplicate the particle buffer. (Perhaps a direct memory access engine could make that copy, leaving the CPU unencumbered.) In contrast, the visual artifacts of rendering the shared particle buffer might be so small and infrequent that the user might not

notice. It's worth trying several variations to find a good compromise between speed and visual quality.

For the fluid simulation to look like a continuous, dense fluid instead of a sparse collection of dots, the sample code uses a lot of tracer particles—tens of thousands, in fact. Arguably, it would look even better if it had millions of particles, but processing and rendering are computationally expensive—both in time and memory. If you used fewer particles of the same size, the rendering would leave gaps. If you increased the particle size, the gaps would close but the fluid would look less wispy—that is, unless the particles grew only along the direction that smaller particles would appear. There are at least three ways to approach this problem:

1. Use volumetric rendering instead of particle rendering. Doing so would involve computing volumetric textures and rendering them with fewer, larger camera-facing quads that access the volumetric texture; the results can look amazing.
2. Elongate tracer particles in the direction they stretch. One way to do that is to consider tracers as pairs, where they are initialized near each other and are rendered as two ends of a capsule instead of treating every particle as an individual blob. You could even couple this with a shader that tracks the previous and current camera transform and introduce a simplistic but effective motion blur; the mathematics are similar for both.
3. Expanding on the idea in option 2, use even more tracers connected in streaks. For example, you could emit tracer particles in sets of four (or some N of your choice) and render those as a ribbon. Note, however, that rendering ribbons can be tricky if the particle cluster “kinks”; it can lead to segments of the ribbon folding such that it has zero area in screen space.

About the Author

Dr. Michael J. Gourlay works at Microsoft as a principal development lead on HoloLens* in the Environment Understanding group. He led the teams that implemented tracking, surface reconstruction, and calibration. He previously worked at Electronic Arts (EA Sports) as the software architect for the Football Sports Business Unit, as a senior lead engineer on *Madden NFL** and original architect of *FranTk** (the engine behind Connected Careers mode), on character physics and *ANT** (the procedural animation system used by EA), on *Mixed Martial Arts**, and as a lead programmer on *NASCAR**. He wrote *Lynx** (the visual effects system used in EA games worldwide) and patented algorithms for interactive, high-bandwidth online applications.

He also developed curricula for and taught at the University of Central Florida, Florida Interactive Entertainment Academy, a top-rated interdisciplinary graduate program that teaches programmers, producers, and artists how to make video games and training simulations.

Prior to joining EA, he performed scientific research using computational fluid dynamics and the world's largest massively parallel supercomputers. Michael received his degrees in physics and philosophy from Georgia Tech and the University of Colorado at Boulder.

Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2016 Intel Corporation