Boeckx Siebe

Creating a fluid simulation in Unreal Engine

Supervisor: Tesch Tom

Coach: Defoort Stephanie

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

Fluid simulations are an ongoing subject, mainly in academic researches. But also in the game industry, water and other fluids of the like are garnering more attention due to the increase of computational power on newer systems. The graphics part of these fluids has definitely seen improvements, but I was wondering why the underwater currents have generally not been implemented yet.

In this paper I will showcase how I implemented a 3-dimensional fluid simulation in Unreal Engine and benchmark the effects it has on performance.

PREFACE

I decided to undertake this subject since fluid interested me greatly but I had no experience with them yet.

The main part of my research was the actual process I followed to achieve my fluid simulation, but if I were to give advice to anyone doing the same, don’t bother creating it in blueprints first. You’ll be much easier of creating it in code from the start. Unreal blueprints are just too convoluted for a complex system that is a fluid simulation.

LIST OF FIGURES

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

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

Figure 1.1: Navier-Stokes equations

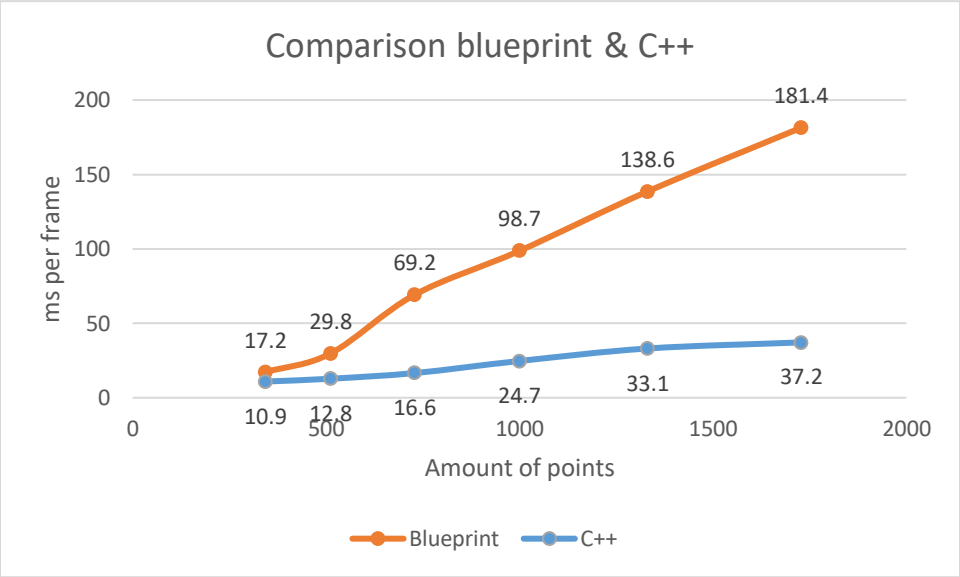


Figure 3.1: Comparison blueprint & C++

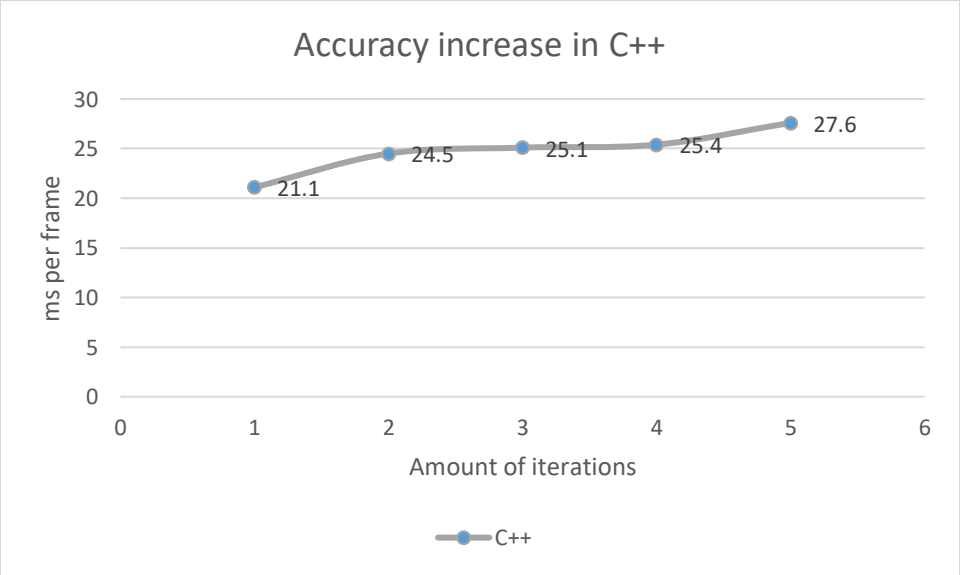


Figure 3.2: Accuracy increase in C++

INTRODUCTION

Fluid simulations have been around for quite some time, starting with Euler, Navier and Stokes back in 1750's to 1850's. These developments led to the **Navier-Stokes Equations**. These are a precise mathematical for almost all fluid flows in nature. These equations are however complex, resulting in only analytical solutions in very simple cases. Progress had thus halted until computers started being used, now algorithms to solve these equations were created but these are fairly complex and time consuming. This is non-optimal for the application in this paper, since we want to be able to theoretically run a game while the simulations run.

To solve this issue, this paper will use the algorithms developed by [1]Jos Stam adding an extra dimension and using more modern coding techniques.

The main question of this paper was, what is the process involved in utilizing real-time fluid simulation to create a lifelike underwater swaying experience?

I expected that the calculations for the simulation would quickly slow down the program even though they would only scale linearly with the amount of grid points.

I was also planning to visualize this simulation by having particle effect get influenced by the fluid simulation, thus showcasing the swaying effect of the simulation.

In this document, references are indicated by numbers, such as [1]Jos Stam. The corresponding numbers can be found at the end of the document in the reference section, along with the full details of each source.

LITERATURE STUDY / THEORETICAL FRAMEWORK

1. FLUID PHYSICS

In mathematics, any given state of a fluid can be modeled as a field of velocity vectors: functions that assign a vector and density to every point in space. These functions are called the Navier-Stokes equations.

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

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

For this application, I used the equations in this form, the variables used are:

- u is the velocity of the point
- ρ is the density/pressure of the point
- ν is the viscosity of the fluid
- κ is the diffusion rate of the fluid
- t is the time
- both f and S are outside inputs, f is added velocity and S added density. You can see f as pushing the fluid and S adding a drop inside of it.
- ∇ is "nabla", it simply indicates change much like δ. In our equations it corresponds to the linear solve function which we will delve into later in the explanation.

The Navier-Stokes Equations provide a detailed depiction of how a velocity field evolves over time, given an initial reference frame. These versions of the Navier-Stokes equations are used for incompressible fluids like water. They would not work if you would try to simulate for example air currents, since air can be compressed. I personally did not research further into the mathematics of these equations since it would be out of the scope of my paper and I do not possess the mathematical background to understand them.

In the real world, fluids change in velocity and density due to billions of miniscule particles colliding with billions of other miniscule particles, billions of times per second. A computer can not perform all these calculations and still perform well enough to run a game, thus we need to create a finite representation of this concept. For this, I will be using a grid with pointVectors at the center of each grid. This pointVector will store the variables for both current- and previous velocities and densities. The calculations for the fluid simulation will be performed by the grid itself, using the data stored in the pointVectors.

Of the two equations, the diffuse function is actually the simpler of the two. It states that the change in density can be described by the three terms on the other side. The first states that the density should follow the velocity field.

The second states that it diffuses at a certain rate and the third says that the density can increase from external sources. Of these, only the first and second need an explanation.

Let us start with the second term. It states that the density will spread if the diffusion rate is larger than 0. For my implementation I assumed that the cells only exchange density with its six direct neighbors. The cell would lose density to its neighbors, but also gain it from them. The rate at which this happens is called the linear solve in my implementation since it provides a linear equation for what was a non-linear one in the original equations. It does this using the [3]Gauss-Seidel relaxation, removing some unnecessary matrix equations. In my project, I called it the LinearSolve function.

Next, the first term states that the densities are effected by the velocity field. In other words, the densities follow the velocities. We could solve this by setting up another linear system using the [3]Gauss-Seidel method, but there is a more efficient way. We could find the particles that would end up at exactly the grid centers after the time step. The amount of density these particles would carry, is simply obtained by linearly interpolating the density at their starting location from the six closest neighbors. To get these position, you could step the grid cells backwards through the velocity field. Afterwards, you interpolate from the previous densities and assign it to the values of the current densities. This is called the AdVect function in my project.

Now that the change in densities has been computed, we can calculate the change in velocities.

Once again we have three terms that describe the evolution of the velocities. The first is self-advection, this may seem obscure, but is simply states that the velocity field moves along itself. The second term is viscous diffusion, describing how the velocity of one point effects and changes the velocity of its neighbors. Finally the third term which is once again input from an external source.

The first and second term can be calculated using the same routines as we used in the density section, this time changing the velocities instead of the densities. But this doesn't produce a realistic flow of fluids. It misses much of the turbulence you would expect. For this, we create a new routine, the Project function. This routine forces the fluid to be mass conserving. This is an important property for fluids that has not yet been enforced in the project.

The Project function is highly mathematical which I didn't get too deep into, since I don't understand most of it. I will however provide the references I used to try to understand it as much as possible. To make sure that the velocity field conserves mass, we use the results of a [4]Hodge decomposition: every velocity field is the sum of a mass conserving field and a gradient field. When you look at the velocity field, we can acquire a height field from it using the [5]Poisson equation. When you subtract the gradient of this height field from our velocity field, you get a mass conserving one. We can reuse [3]Gauss-Seidel in the [5]Poisson equation.

That summarizes how this fluid simulation works. Most of my information on how to create the functions came from [1]Jos Stam so I highly recommend you read through that paper before attempting a fluid simulation like this.

RESEARCH

The first part of the research done for this paper was the description of the actual process I took to implement the fluid simulation into an Unreal Engine project.

The second and smaller part consists of the benchmarking I did, the first benchmark compares the blueprint application to the C++ version and the second investigates the effect that increasing the accuracy of the simulation has.

2. CREATION PROCESS

2.1. BLUEPRINT

Generally, working with Unreal Engine blueprints quickly becomes convoluted, and that was most definitely the case this time too. You would usually only use blueprints for small calculations or interacting with Unreal Engine implementations like the Niagara particle system, but I made the mistake of trying to make the entire fluid simulation in blueprints first to quickly prototype and test the implementation.

I started with creating point vectors containing all the variables like the velocity vector and density and a grid manager that would create and store these objects. Creating and populating the grid was rather easy, keeping in mind that I'd need an extra layer of points in each direction to account for the boundary calculations.

The next step, following the paper from [1]Jos Stam, was to add the density functions. Creating the linear solver was pretty straightforward since the functionality itself isn't so complicated, what I didn't realize at the time is that that function would only be applicable for the linear solving of the densities of the simulation. But with that, the first part of the fluid simulation was completed.

The process started getting difficult when I started on the velocity solver part of the system. Here I realized that I couldn't reuse my linear solver like [1]Jos Stam and [2]Mike Ash did in their simulations and thus had to make a separate one for each variable. Also because the linear solve function was already so big, I had a lot of troubles correctly making one to change the velocity instead of the density. When I finally had that part working it was time to make the project function but this caused for even more issues. At first I was using the same variable changing as happened in the reference algorithms, but bugs kept appearing and the simulation always exploded. I eventually decided to just store the divergence and pressure as separate variables instead of storing them in the previous velocities. This solved the simulation exploding but a lot of time had passed.

The last step was fixing the boundary issues, once again I had to make different functions for each variable. The only positive aspect of this was that I had no need for the enum used in the algorithms, since that's how they differentiated which variable was being changed.

The entire process was far from quick, since I used a point vector struct containing the different variables per point instead of storing all the variables as arrays in the grid manager, my data accessing behavior was substantially different from that of the algorithms of [1]Jos Stam and [2]Mike Ash. Add to this the node system that makes even a simple algebraic equation look intimidating and you have huge blueprints for what would be a couple lines of code.

Another effect that the different data structure brought with it is it removed the reusability of the functions used in the original algorithm. I tried implementing that reusability, but the resulting functions were too inelegant to use, so I fell back to making each function for every variable that needs to use them. This resulted in many, extremely similar functions making the entire system even harder to maintain.

The final step was creating two Niagara particles systems, one to visualize the velocities and the other to show the density of the fluid. The first was rather simple, it just spawns little floating particles that have their velocity set by the point vector that spawned them. The second effect was a little harder, I wanted it to look like a drop of ink spreading but that was a little too hard. I settled for an effect that increases in size with higher density so you can see the density diffuse throughout the fluid.

2.2. C++

This process went much smoother than creating the blueprint version. There were however a few caveats in this process as well.

For example, at the start I had a slight issue correctly spawning the actors at their intended location.

Throughout writing the C++ version of the simulation I came to realize that this was much quicker and efficient than creating it in blueprints. There were a few instances where I struggled to identify bugs caused by minor typos, but one particular bug proved to be challenging to pinpoint. Due to this bug inside the project function, the velocities kept increasing exponentially until eventually the program crashed. It was due to the x and y values changing inside the previous velocities and not getting properly reset. This resetting process was simple to implement, it only confuses me why I didn't have to do this in the blueprint version and why [1]Jos Stam and [2]Mike Ash didn't have to implement this.

Nothing had to be changed regarding how the particles moved and interacted since I was using the same structure as in the blueprint version. I simply added and linked up the same particle effects I used in the blueprint application.

2.3. REFLECTION

Looking back on the process, I was wrong to create the simulation in blueprints first. This took a lot of time for what was eventually done in C++ in a couple of days. Although I succeeded in having particles react to the changing values in the simulation, it was much better visible in the C++ version due to the higher performance, I will cover this in the next section. Both applications do their calculations once per frame, but this also means that the project has to wait for these calculations before rendering a new frame. The simulations themselves are not framerate dependent as they take delta time into account. Lower framerate however result in the simulation seeming to slow down. It causes the velocities in the fluid to appear slower and the diffusion taking longer than usual.

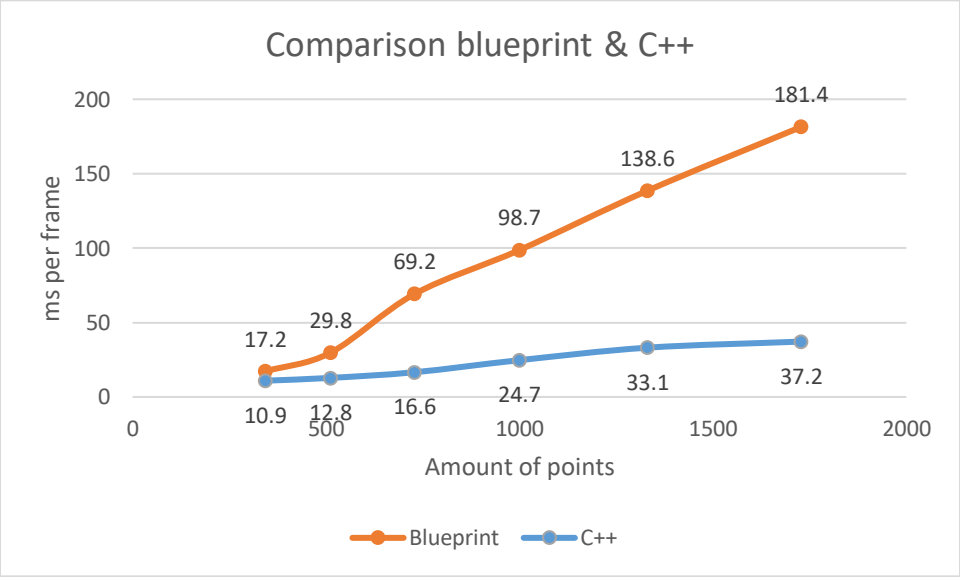
3. BENCHMARKING

3.1. BLUEPRINT VS C++

The first performance test I did was also the one that interested me most. How much more efficient was the C++ application compared to the same project but in blueprints. I performed 5 comparisons, changing the size of the grid with each one and calculating the average microseconds(ms) per render frame. The accuracy of all the simulations stayed constant with 4 iterations being used in the linear solve functions. The amount of points include the boundary points, so for example an 8x8x8 grid becomes a 10x10x10 grid with the boundaries, resulting in 1000 points. The data was gathered by using unreal insights while running a development build of each project separately. There were other applications running on my computer at the same time, but these didn't change during the entire process so it should have a minimal effect on the results gathered.

The microseconds shown in the graph are the averages calculated after letting each build run for between 20 and 30 seconds.

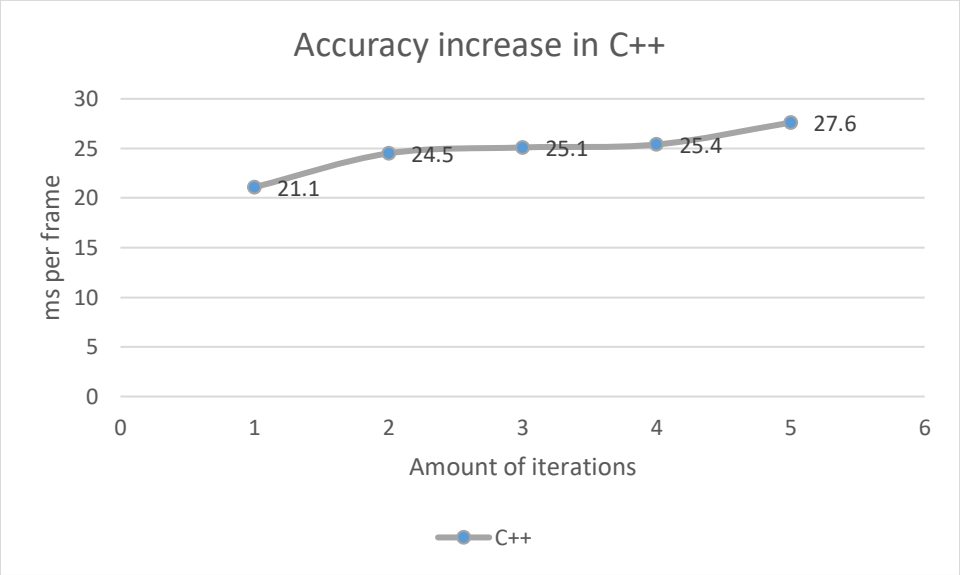
The data showed that the C++ application was always more efficient in simulating the fluids, increasing it's advantage over the blueprint version when the amount of calculations increased.



Another interesting result to note is that in both cases, the time it took for the simulation to perform all the calculations increases linearly as expected. The linearity of all functions, which are essentially linear equations, is the reason behind this phenomenon. In simpler terms, this demonstrates that the simulation grows in accordance with O(N).

3.2. ACCURACY INCREASE

The second test I did on my project was to see how increasing the accuracy of the simulation affected the time it took for each frame. The results here were what I expected again, the duration increased linearly with the amount of iterations performed in the linear solve function. This function is what dictates the accuracy of the simulation here increasing the iterations increases the accuracy. The data was gathered in the C++ version of the simulation with 1000 simulated points, or an 8x8x8 (10x10x10) grid.



DISCUSSION

My main point for discussion is the change from the original non-linear Navier-Stokes equations to the linear implementations used inside my project. This results in my simulation to not be completely mathematically correct, but it does produce realistic results at a faster rate. That way I can run the program and still move around and see particles being moved in real time.

Another point for discussion are the changes I brought from [1]Jos Stam's implementation and if they are actually beneficial. I added another dimension to the simulation, much like [2]Mike Ash, but also changed how the values were stored by creating a secondary component. Namely the pointVector. This caused for me to make a lot of incredibly similar functions, with their only change being which variable got adjusted. This makes my code convoluted compared to the referenced works.

CONCLUSION

In conclusion, it is possible to make a real time fluid simulation in unreal and have objects interact with it. The main downside are the costly calculations so, at least my version, is not realistic to implement in a game where other performant interactions need to take place. The performance requirements to run a simple, even rather small scale simulation just far outweigh the joy a player could get from such an accurate simulation.

FUTURE WORK

One extension to this project is one where I lack the knowledge, but you could implement a better visualization of the fluid. This applies mainly to the diffusion visuals. I do not have the skillset to make the diffusion look like an actual drop of ink spreading throughout the fluid.

You could combine this with creating actual input into the simulation, either through vents constantly spewing new density and velocity, or through direct input from the user.

Another possible extensions to this project could be creating blocking objects like rocks and boulders that the fluid cannot pass through. This could perhaps be achieved by adding the pointVectors that fall inside the structure to the boundary equations.

Additionally, an extension to this simulation could be creating objects like seaweed that have different sections move in different directions according to the values produced by the simulation. This could then create a waving effect on the plant similar to what happens in real life.

A final extension I could think of and a much simpler one to implement, is a character that when inside the fluid has its velocity adjusted according to the simulation. This would make it sway much like the particle effects do already.

CRITICAL REFLECTION

During the creation of this paper, I realized that I did not have enough background knowledge to fully understand how a fluid simulation works. But I found a way to work through that, I only tried to learn about what is most essential to the simulation instead of also researching about every mathematical equation used like I would usually do.

I also deepened my understanding of Unreal Engine through the creation of the project, learning more about the intrinsic workings of objects, the differences between blueprints and code and the performance impacts of various entities like particles.

REFERENCES

Coding Challenge #132: Fluid Simulation, 2019. https://www.youtube.com/watch?v=alhpH6ECFvQ.

'Fluid Simulation for Computer Animation'. Accessed 9 November 2023. https://www.cs.ubc.ca/~rbridson/fluidsimulation/.

Gabriel Weymouth. 'Lily-Pad: Real-Time Two-Dimensional Fluid Dynamics Simulations in Processing.' Accessed 26 October 2023. https://github.com/weymouth/lily-pad.

[3]'Gauss–Seidel Method'. In *Wikipedia*, 12 January 2024. https://en.wikipedia.org/w/index.php?title=Gauss%E2%80%93Seidel_method&oldid=1195077946.

[4]'Hodge Theory'. In *Wikipedia*, 8 January 2024. https://en.wikipedia.org/w/index.php?title=Hodge_theory&oldid=1194422053.

[1]Jos Stam. 'Real-Time Fluid Dynamics for Games', n.d., 17.

'Navier–Stokes Equations'. In *Wikipedia*, 3 November 2023. https://en.wikipedia.org/w/index.php?title=Navier%E2%80%93Stokes_equations&oldid=1183363016.

Memo Akten | Mehmet Selim Akten | The Mega Super Awesome Visuals Company. 'MSAFluid (2008)'. Accessed 26 October 2023. https://www.memo.tv/works/msafluid/.

[2]'Mikeash.Com: Fluid Simulation for Dummies'. Accessed 4/01/2024. https://mikeash.com/pyblog/fluid-simulation-for-dummies.html.

OpenAI. 2022. "ChatGPT Documentation." OpenAI. https://chat.openai.com.

[5]'Poisson's Equation'. In *Wikipedia*, 22 October 2023. https://en.wikipedia.org/w/index.php?title=Poisson%27s_equation&oldid=1181351259.

Unreal Engine City Building Game - The Grid - EP 5, 2021. https://www.youtube.com/watch?v=DwPHsLJv5Gw.

Unreal Engine City Building Game - The Grid Advanced, Neighbouring Cells - EP 6, 2021. https://www.youtube.com/watch?v=k_LIXDfOxD8.

'Water System'. Accessed 26 October 2023. https://docs.unrealengine.com/5.3/en-US/water-system-in-unreal-engine/.

ACKNOWLEDGEMENTS

I would like to first and foremost thank ChatGPT for helping me write this paper. I had little experience with writing a formal document like this beforehand so it provided significant assistance to me.

Secondly I'd like to thank Defoort Stephanie for the couching she did for me and my colleagues.

Last but not least, I'd like to thank Tesch Tom for being the amazing tutor that he is.

APPENDICES

I will leave my project files online on OneDrive. The Unreal Insight data and builds used during benchmarking will also be stored there for those interested.

Siebe_Boeckx_Graduation_Work_23-24