

Real-Time Multiple Fluid Simulation for Games

Jacob Worgan (s5107963)

27th October 2022

Abstract

Fluid simulation is a well studied and important aspect of computer graphics, and with increasing hardware power they are becoming more common in interactive applications. This thesis focusses on exploring the idea of how these fluid simulations can be used in the context of player interaction, and the novel gameplay that can arise from the increasing usage of these simulated elements.

Contents

1	Introduction	3
2	Previous Work	4
2.1	Niagara Fluid Plugin	5
2.2	Fluid Ninja	5
3	Technical Background	7
3.1	Grid-Based Eulerian Fluid Simulation	7
3.1.1	Fluid Diffusion	8
3.1.2	Advection	8
3.2	Particle-Based Lagrangian Fluid Simulation	8
3.2.1	Density	8
3.2.2	Pressure	9
3.2.3	Viscosity	9
3.2.4	Surface Tension	9
3.2.5	Extensions for Multiple Fluids	10
3.2.6	Smoothing Kernels	10
4	Solution	11
4.1	Player Character	11
4.1.1	Absorbtion & Releasing	11
4.2	Grid-Based Fluid Simulation	12
4.3	Importing & Exporting Data to Niagara Effects	13
4.3.1	Exporting Data from Niagara	13
4.3.2	Absorbtion & Releasing Calculations	13
4.3.3	Importing Data to Niagara	14
4.4	Particle-Based Fluid Simulation	14
4.5	Extras	14
5	Conclusion	15
5.1	Future Work	15
	Bibliography	15

List of Figures

2.1	A selection of the template fluid systems included in the Niagara Fluid Plugin	6
4.1	The sponge material at minimum and maximum absorption, showing the difference in the wetness parameter.	12
4.2	The Export Particle Node	13

Chapter 1

Introduction

Fluid simulation is a complex field of study in the realm of computer graphics, with many differing approaches being continuously developed that consider a wide variety of different use case requirements. Fluids show complex behaviours and significantly add to the 'richness of virtual worlds' [Muller et al.(2005)]. However, in real-time applications, and especially in videogames, there has been a dirth of fluid simulations that allow for interaction outside of collision. As computer hardware increases power, fluid simulations potentially offer a wide variety of novel interactions, and it was in the potential of these interactions that was the catalyst for this master's thesis.

To limit scope, a single core gameplay concept was developed, that being of the player controlling a sponge character that has the ability to absorb and release water at various points around the game world. Fluids could then interact with eachother, or with various other objects in different ways, taking inspiration from common chemistry experiments.

This core gameplay mechanic allowed for an exploration into various forms of fluid simulation, while narrowing scope of research towards a specific end use case.

For clarity on part of the reader, in various areas of the project this concept has been refered to by a working title of 'Frankensponge'.

Chapter 2

Previous Work

With the ubiquitous frequency of fluids present in everyday life, they are a common sight in videogames, and over the years a variety of techniques of representing them have been developed and used.

Most common is the use of computationally inexpensive billboards and meshes, covered with sometimes dynamic, orthertimes flipbook-animated materials. These are, naturally, not simulated, and are therefore relatively static. They work for many situations, but are limited in terms of possible player interaction.

A good recent example, however, of a more interactive approach to rendering fluids is in the game Half-Life Alyx [Valve(2020)]. Here, shaders are used to give the impression of bottles of liquid without having to do a full fluid simulation [Polygon(2021)]. Again this is much cheaper in terms of computational cost, but it still provides a greater level of realism and player interactivity as the shader responds to player input, with the liquid appearing to move about as the player tilts and shakes the bottle in VR. This being a VR game is also important; these new mediums alongside the increase in hardware allow for (and in some cases makes it a lot more necessary to have) new levels of player interaction that can provide new gameplay and realism.

When looking at full fluid simulations, however, these are much less frequent due to the high cost, however a number game engines currently include fluid simulation systems, and in Unreal Engine there are two main ones that are of particular relevance to this project. The first is a plugin called 'Niagara Fluids' [Epic2020] included as part of the release of Unreal Engine 5.0, and the second a set of plugins available on the Unreal Marketplace called 'FluidNinja' and 'FluidNinja Live' [Andras(2022)], the former being a tool to bake out fluid simulations for flipbook animations for effects and materials and the latter a number of real-time fluid simulation effects.


2.1 Niagara Fluid Plugin


The Niagara Fluids Plugin [Epic2020] is a plugin that ships with Unreal Engine 5, and provides a number of methods and templates relating to 2D and 3D single simulations of both liquid and gasiform fluids. These simulations use a hybrid approach with the simulation taking place on particles that are driven in part by grids for attributes such as velocity, pressure, and divergence. As with most sample content by unreal they are used as reference and as a base for more complex simulations, and as such are relatively simple in appearance and interaction, with few allowing for even collision with outside components.


2.2 Fluid Ninja


Fluid Ninja is an add on for Unreal Engines 4 and 5, with two major versions being FluidNinja that is used to develop and simulate fluids to be baked into flipbooks for low-cost fluid effects, and FluidNinja Live which implements a wide range of real-time fluid simulation effects, including smoke, fire, and liquids [Andras(2022)]. A lot of the simulations take place using grid-based methods, with particular focus on aesthetics and optimisation for speed. There are a few examples of interaction, mainly with the simulations being affected by object collisions.

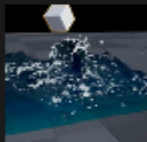
Figure 2.1: A selection of the template fluid systems included in the Niagara Fluid Plugin

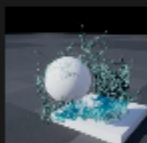
- ▶ 2D Gas
- ▼ 2D Liquid
 - 

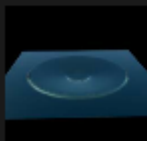
Grid 2D FLIP Hose
Constant emission hose with 2d liquids in a closed container
 - 


Grid 2D FLIP Pool
Pool of water in a closed container
 - 

Grid 2D FLIP Splash
Single water splash
- ▶ 3D Gas
- ▼ 3D Liquid
 - 

Grid 3D FLIP Hose
A liquid system with a pool of static water that also has constant emission of new particles.
 - 

Grid 3D FLIP Pool
A liquid system with a pool of static water. Static meshes will act as collisions and cause splashes if they are moving.
 - 

Grid 3D Flip Splash
A liquid system with a single splash burst of water
- ▼ Shallow Water
 - 

Grid 2D SW Drop
Simple shallow water system
 - 

Grid 2D SW Particle Collisions
Particles colliding with shallow water

Chapter 3

Technical Background

Fluid: "A substance that cannot support shear stress in static equilibrium" or "a substance that flows because it cannot resist deformation". [Muller et al.(2005)]

In the domain of fluid simulation there are a great number of different techniques developed that are suitable to different contexts and requirements, however, like for most simulated phenomena, there are only a couple main approaches of simulating that this variety of techniques builds off of. In this case, there is the grid-based Eulerian approach, and the particle-based Lagrangian approach. Both techniques as a base incorporate and build from the Navier-Stokes equations developed in 1822 as a method of describing the dynamics of fluids [Muller2003].

$$\nabla \cdot u = 0$$

$$\rho \frac{Du}{Dt} = -\nabla \mu + \nabla^2 u + \rho F$$

3.1 Grid-Based Eulerian Fluid Simulation

The grid-based Eulerian fluid simulation method was well summarised by [Stam(2003)].

As the name implies, the grid-based method simulates a fluid in a finite region by dividing the simulation area into identical cells that are then sampled into at the given cell's centre.

This approach uses a variation on the Navier-Stokes equations to model a fluid's density moving through a velocity field:

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

This equation gives three terms, the first, $-(u \cdot \nabla) \rho$, moves the density along the velocity field; the second, $\kappa \nabla^2 \rho$, diffuses the density at a given rate; and the final term, S , is for inputs that increase density with new fluid coming from a given source.

3.1.1 Fluid Diffusion

Diffusion calculations exchange density with directly surrounding neighbour cells, at a given diffusion rate. An unstable version is to model this forward:

```
Cell Density = (cell diffusion rate * value of
surrounding cells) - (4 * value of current cell);
```

In this version values will oscillate and diverge, so a more stable version goes backwards to find where the density of a given cell is coming from:

```
Cell Density = (value of current cell + cell diffusion
rate * value of surrounding cells) / (1 + 4 * cell
diffusion rate);
```

This version can also handle any delta time value without the simulation breaking down.

3.1.2 Advection

Similar to the diffusion, a simple linear backtrace works backwards to find the cells that feed new density values into the current cell, rather than working the current cell's values forward along the advection velocity field. The density values of the four cells around the backtraced point are then lerped to find the new density value.

3.2 Particle-Based Lagrangian Fluid Simulation

The particle-based Lagrangian fluid simulation method was summarised by [Muller2003]. The specific techniques are built on the ideas of 'Smooth Particle Hydrodynamics', a particle method that does not need a grid to calculate spacial derivatives, instead being calculated through a series of interpolation formulae [Monaghan(1992)].

Here, fluids are represented by a set of positions, masses, and additional attributes, computed as a smooth, continuous field from discrete values sampled at particle locations through the equations from Smooth Particle Hydrodynamic (SPH) methods [Muller et al.(2005)].

3.2.1 Density

The first attribute that is calculated is density:

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W(r - r_j, h) = \sum_j m_j W(r - r_j, h)$$

Here, the output density ρ_S is driven by a sum of the masses of the other particles m_j , weighted using a smoothing kernel $W(r, h)$ weighted by the distance between the current particle r and the current summed particle r_j .

3.2.2 Pressure

The application of the SPH rule to the pressure term of the Navier-Stokes equations $-\nabla p$, and made symmetrical, yields:

$$f_i^{pressure} = - \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r - r_j, h)$$

Here, the total pressure for the current particle i is calculated by the sum of the masses of the other particles m_j and the individual pressures p , calculated using the ideal gas state equation:

$$p = k(\rho - \rho_0)$$

Where k is the temperature-dependent gas constant, or stiffness of the fluid, ρ is the density calculated in the previous section, and ρ_0 is the rest density of the fluid. The full pressure calculation then is weighted for each particle summed using a smoothing kernel, as per the SPH method.

3.2.3 Viscosity

Particle viscosity is used to accelerate particles in the direction of the relative speed of the other particles in the surrounding environment. Using the SPH rule applied to the viscosity term $\mu \nabla^2$ and symmetrising the resulting formula gives:

$$f_i^{viscosity} = \mu \sum_j m_j \frac{(v_j - v_i)}{\rho_j} \nabla^2 W(r - r_j, h)$$

Where μ is the particle viscosity.

3.2.4 Surface Tension

Particles in a fluid are subject to attractive forces from neighbouring particles; intermolecular forces inside a fluid are equal, however on the molecular level they are unbalanced at the free surface (boundary between fluids). These net forces, including surface tension, act in the direction of the surface normal towards the fluid. A tension coefficient forms the strength of this surface tension force, dependent of the attributes of the two fluids that form the surface.

With pressure, viscosity, and surface tension all calculated, the particles can then be accelerated with the combined forces of these parameters with the following formula:

$$a_i = \frac{1}{\rho_i (f_i^{pressure} + f_i^{viscosity} + f_i^{external})}$$

Where $f_i^{external}$ is the sum of all external forces, including gravity and surface tension.

3.2.5 Extensions for Multiple Fluids

In order to extend this simulation to include multiple fluids, [Muller et al.(2005)] suggests storing what would otherwise be global variables shared by all particles, such as viscosity or the gas constant, and storing them as particle attributes that can then be changed per particle. This gives way to several changed formulae from those above. The viscosity vector calculation needs to support separate viscosity values between the two particles being compared in the sum:

$$f_i^{viscosity} = \sum_j \frac{\mu_j + \mu_i}{2} m_j \frac{(v_j - v_i)}{\rho_j} \nabla^2 W(r - r_j, h)$$

Other suggested aspects are relevant to simulate specific phenomena within the fluid, such as air pockets with dynamic air particle creation and deletion which are not relevant to the requirements of this project.

3.2.6 Smoothing Kernels

The smoothing kernels proposed include a general case kernel:

$$W_{poly6}(r, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & : 0 \leq r \leq h \\ 0 & : otherwise \end{cases}$$

A kernel for pressure calculation:

$$W_{spiky}(r, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & : 0 \leq r \leq h \\ 0 & : otherwise \end{cases}$$

And a kernel for viscosity calculations:

$$W_{viscosity}(r, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & : 0 \leq r \leq h \\ 0 & : otherwise \end{cases}$$

$$\nabla^2 W(r, h) = \frac{45}{\pi h^6} (h - r)$$

$$W(|r| = h, h) = 0$$

$$\nabla W(|r| = h, h) = 0$$

Chapter 4

Solution

Starting the solution phase of the project, Unreal Engine 5 was chosen as the engine for the project in order to use the robust Niagara effects system.

4.1 Player Character

Given that this is a game, creation started with the player character and its controls. Inheriting from the base Unreal Engine class of `ACharacter` was the `APlayerChar` class, which uses the built in player movement and navigations systems of Unreal Engine. The player can move across a 2D plane, with 2D being chosen to limit the simulations similarly to 2D to reduce overall complexity of the required solution to the essential elements. The player can also jump and crouch, and absorb and release fluid.

4.1.1 Absorbtion & Releasing

Two inputs were created to control the player's ability to absorb and release liquid, which set publicly readable booleans that control importing and exporting of fluid particle data described in following sections. .

As the player absorbs more, a few adjustments are made to the `CharacterMovementComponent` that controls player movement, based on the current amount the character has absorbed as a percentage based on the total amount it can absorb (a value that is exposed as editable in blueprints and on instanced versions of the character). Jump velocity and ground friction are both decreased, to replicate a slippier and heavier character. Also, a dynamic material instance is used as the material on the sponge character (called 'M_Frankensponge') with a parameter called 'wetness' that increases specularity and decreases roughness parameters on the material when increased. This gives the sponge a shinier appearance, as if wet. Initially this material instance was set in C++ code, however creating the material this way caused issues with saving scenes in which the player pawn was present, as the dynamically created material was not saved to

a file, so this was moved to a blueprints version of the character which inherited from the C++ version of the player character actor.

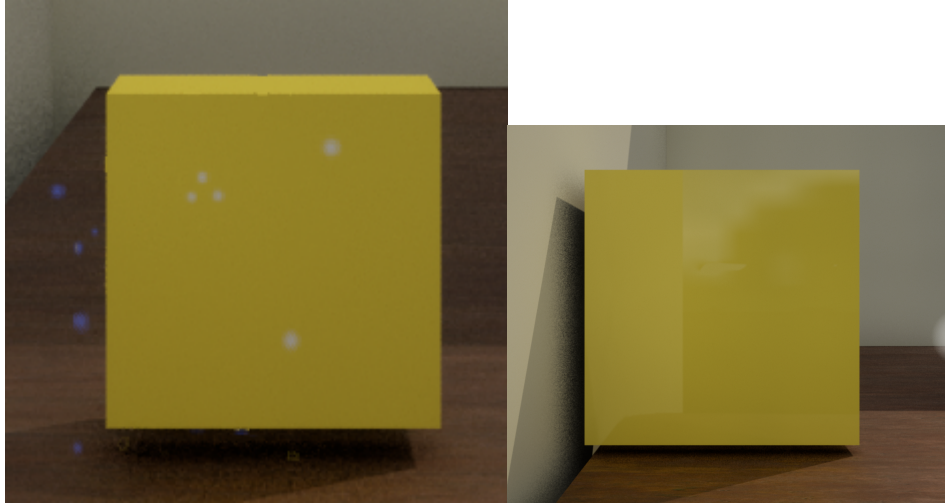


Figure 4.1: The sponge material at minimum and maximum absorption, showing the difference in the wetness parameter.

4.2 Grid-Based Fluid Simulation

With the ability to import and export data to the Niagara effect, work could then be done on making a fluid simulation that could use this data. The grid-based method was chosen to test first due to it being closer to the techniques used by the example fluid simulations given by Epic in the Niagara Fluid plugin.

Much of this simulation was achieved through transcribing equations and code given in the existing papers given in the Technical Background section.

Grid2DCollections were used as the data storage method, allowing for multiple layers of data shared across all points on the 2D simulation grid. Data from this could then be directly mapped to a Render Target used in rendering the simulation to a material.

Based on [Stam(2003)], the order of simulation goes from fluid diffusion, to fluid advection along the vector grid, with two copies of the fluid grid, one current to work on and one to hold the previous information, with the two being swapped after each simulation stage to clear the working grid and move information to the previous grid. All are given the same resolution of cells in the X and Y by use of system parameters set on the overview node.

Diffusion and Advection calculations were carried out in two scratchpad modules using Custom HLSL nodes that allow for the direct writing of HLSL for calculations, which meant that several dozen nodes with dozens of connections

could be reduced to a dozen lines of code which was much more manageable and much easier to parse for debugging purposes.

4.3 Importing & Exporting Data to Niagara Effects

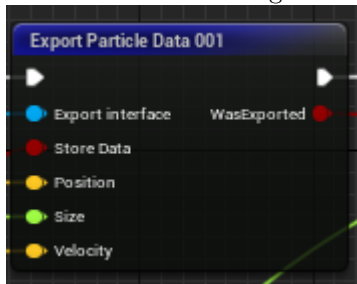
With an initial fluid simulation completed, work could then be done on the ability to import and export data to this Niagara effect to power the simulation. This was a key technical challenge to allow for the core gameplay ability of being able to absorb and release liquid, to transport it between areas of the game world.

In the context of Unreal Engine's Niagara effects system, this importing and exporting is achieved with the use of User Exposed parameters on the Niagara end, and a set of function calls on the C++/Blueprints end.

4.3.1 Exporting Data from Niagara

Exporting data from Niagara is done through an 'Export Particle Data' node on the node graph, or, at the emitter level, using the 'Export Particle Data to Blueprint' module. As can be seen in the figure below, exporting in this way allows the user to export a total of three parameters, of two vectors and a float, for a total of 7 float values maximum per particle. Although labelled, they can export any arbitrary particle data of the given size, however for the case of exporting fluid particle data only the Position and Size values were required.

Figure 4.2: The Export Particle Node



Once exported, this calls functions relating to a callback handler, 'INiagaraParticleCallbackHandler', which is a function that is called from the Niagara system with the exported data as its given input parameter.

4.3.2 Absorbition & Releasing Calculations

To achieve the absorbition and releasing actions key to the gameplay element of absorbing/releasing fluid, at this point between exporting and importing data

to the Niagara system collision checks are carried out between the player character's bounding hitbox and each of the exported particles.

Collision is carried out by a separate function, `FindPlayerBoundedParticles`, which tests the player `BoundingBox` boxcomponent against a sphere at each particle location of the given particle's size.

If the player is absorbing or releasing, checks are performed to find out how much can be absorbed/released (based on how much the player has absorbed, the max amount it can absorb, and the absorption/release rates), and this is split among all the particles that were overlapped to

4.3.3 Importing Data to Niagara

Importing data back into Niagara is simpler on the C++/Blueprints side, with a set of functions for importing different types of values (float, vector, etc.), with array and non-array versions.

In the Niagara effect, a scratchpad module goes through for each particle and reads in the new particle size.

4.4 Particle-Based Fluid Simulation

After producing a test of the grid-based fluid simulation method, another Niagara effect was created to produce a particle-based fluid simulation, again largely achieved by transcribing the equations explained in the Technical Background section in scratchpad modules as part of separate simulation stages, utilising Custom HLSL nodes. Extra attributes were added to particles, as per the multiple fluid simulations of [Muller et al.(2005)], including particle specific gas constants, viscosities, and rest densities, allowing for different fluid types to be simulated together in the same particle system.

4.5 Extras

There are a few last parts that were also completed as part of this project.

A user interface was created to display the percent to which the player character is full of liquid.

A range of tests were completed to evaluate the integrity of C++ classes. Testing is not always feasible for very visual simulated projects such as this, especially when using an effects system like Niagara, however tests were written to make sure actors have necessary values.

A 'water spout' actor was also created, with a simple fountain effect and a hitbox that if the player is inside can use to absorb an unlimited amount of water, for testing purposes and potentially for gameplay purposes in a hypothetical feature complete version of the game.

Chapter 5

Conclusion

The overall system still needs further refinement to make the visual aspects of the project look more appealing, however all the requirements of the system are, at least individually, fulfilled.

Unfortunately, documentation for many of the aspects of Unreal Engine that were used throughout this project was limited or in many cases absent, which was a major challenge. This was especially true for the HLSL scripting that is possible through HLSL script nodes in Niagara effects; while HLSL is documented, unreal expands with its own functionality that is hard to find.

Importing data into a Niagara effect is relatively simple, as it is possible to import unlimited arbitrary data, however to export data is limited. Render targets can be exported, but are slow and complex to code or use in blueprints, and the only alternative is the particle export which has limits to 7 floats per particle, and data has to be attached to a particle to export, limiting useability.

5.1 Future Work

To build off of these simulations there is the potential to increase the number of elements that interact with the fluids, given that this project includes data importing and exporting to Niagara systems. These could include sources of heat that increase temperature of particles, or trapped particles that increase in pressure to push external objects, rather than it just being the fluid simulation that reacts to the collision of other objects.

Bibliography

- [Andras(2022)] Ketzer Andras. 2022. FluidNinja. <https://drive.google.com/file/d/1I4dglPjeXLcNkSGxGok8sQCy59qgYcF9/edit>
- [Foster and Metaxas(1996)] Nick Foster and Dimitri Metaxas. 1996. Realistic Animation of Liquids. *Graphical Models and Image Processing* 58 (9 1996), 471–483. Issue 5. <https://doi.org/10.1006/GMIP.1996.0039>
- [Games(2022)] Epic Games. 2022. Niagara Fluids Reference Guide. <https://docs.unrealengine.com/5.0/en-US/niagara-fluids-reference-in-unreal-engine/>
- [Monaghan(1992)] J.J. Monaghan. 1992. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics* 30 (1992), 543–574.
- [Muller et al.(2005)] Matthias Muller, Barbara Solenthaler, Richard Keiser, and Markus Gross. 2005. Particle-Based Fluid-Fluid Interaction. *SIG-GRAPH/Eurographics Symposium on Computer Animation*, 237–244. <https://dl.acm.org/doi/10.1145/1073368.1073402>
- [Polygon(2021)] Polygon. 2021. Why the liquids in Half-Life: Alyx look so dang good.
- [Stam(2003)] Jos Stam. 2003. Real-Time Fluid Dynamics for Games. *Game Developer’s Conference*.
- [Stora et al.(1999)] Dan Stora, Pierre-Olivier Agliati, Marie-Paule Cani, Fabrice Neyret, Jean-Dominique Gascuel, Jean-Dominique Gascuel An, and Dan Stora Pierre-Olivier Agliati Marie-Paule Cani Fabrice Neyret Jean-Dominique Gascuel. 1999. Animating Lava Flows. , 203-210 pages. <https://hal.inria.fr/inria-00510066>
- [Valve(2020)] Valve. 2020. Half Life: Alyx.
- [Yang et al.(2015)] Tao Yang, Jian Chang, Bo Ren, Ming C. Lin, Jian Jun Zhang, and Shi Min Hu. 2015. Fast multiple-fluid simulation using Helmholtz free energy. *ACM Transactions on Graphics* 34. Issue 6. <https://doi.org/10.1145/2816795.2818117>