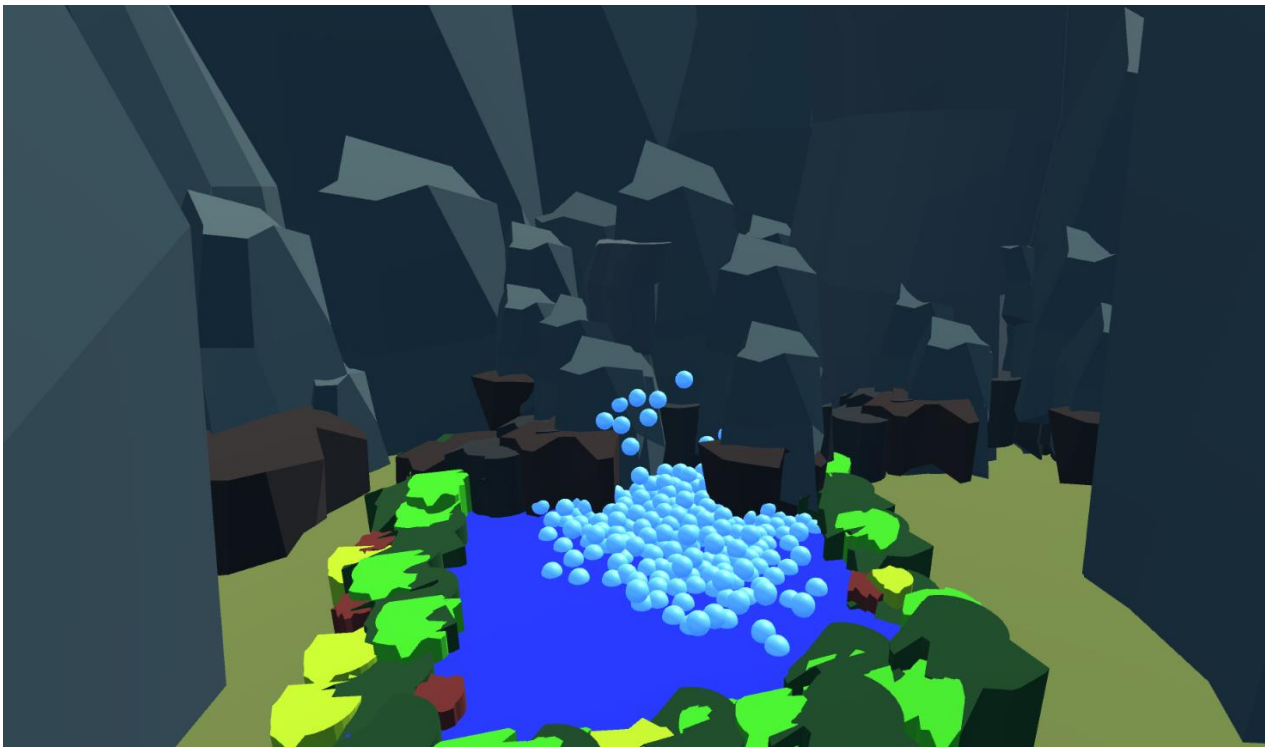
# Creative Technologies Project: Fluid Dynamics Simulation Using Smoothed Particle Hydrodynamics

**Kristian-Alekzandar Bakov**
kristian2.bakov@live.uwe.ac.uk
Supervisor: Tom Bashford-Rogers

**Department of Computer Science and Creative Technology**
University of the West of England
Coldharbour Lane
Bristol BS16 1QY



**Abstract**

This project has researched multiple methods of fluid simulation and their application in real-time game environments with strong emphasis on performance. The methods researched are particle and grid based, as well as combinations between the two. The outcome decision is to utilize the flexibility of a particle based Smoothed Particles Hydrodynamics method, inspired by Kim, D., 2016, inside of Unity game engine. To test performance, different approaches to implementing the method are compared and the outcomes evaluated.

**Keywords**: Smoothed Particle Hydrodynamics, Fluid dynamics, Game environments, Unity game engine.

**Brief biography**

I am a third-year undergraduate student at the University of the West of England on a BSc Games Technology course. This project is a result of my interest in real-time simulations and their application in games. I chose to do a physics simulation because I am passionate about making complex systems for game engines and want to undergo a challenging task which establishes a strong portfolio piece that can be used for future job applications.

**How to access the project**

Full project source code and standalone executables are available at:

https://github.com/KristianBakov/SPHFluidSimulationCTP.git

The project video link is available in the GitHub readme.

## 1. Introduction

Movies, animations, cinematics, and games use fluid simulation to represent natural bodies of water and liquids, but while the former may benefit from having unlimited time to render scenes, games need a solution that is optimal to run in real-time. Games require many uses of fluid dynamics solvers, from fluid simulation to special effects. Papers outline different methods of achieving optimal fluid solver solutions (Gourlay, M., 2012 and Harris M. 2007), proving the most optimal solution depends on the use scenario.

The aim of this project is to develop a real-time fluid simulation suitable for use in game environments made in Unity game engine. In the research report, several different methods of simulation were considered with the final choice being the particle-based method Smoothed Particle Hydrodynamics (SPH). Once SPH was chosen, the project focused on improving performance and quality of the simulation so it can be applied to a game environment.

SPH was initially developed for astrophysical problem solving but is now often used to simulate how fluid behaves (Lucy, B., 1977, Monaghan, J. and Gingold, A., 1983). It is a boundless Lagrangian method, which means that it consists of multiple entities (or particles) that carry individual values as well as global fluid body values. It is used when rendering fluid in movies, cinematics, and games.

The three main aspects of a SPH fluid simulation that need to be balanced are:

- Accuracy
- Performance (optimization)
- Visuals

The project focuses on delivering a high accuracy simulation while keeping performance and visuals at the highest possible level through different optimization methods, such as importing a C++ plugin for faster compilation time and implementing a Spatial Partitioning System (Spatial Hashing).

The project development has revolved around the questions:

- Why SPH and how does it work?

- How to apply SPH to a game environment effectively and efficiently?

- How does Unity's pipeline work and how does that affect the project's outcome?

The final implementation of the SPH solver achieves a simulation that consists of 300 fluid particles at a steady 60 frames per second. After using spatial hashing, the same solver can handle up to 450 particles, and after implementing the code though a C++ plugin with the spatial hashing, that number rises to 1000 particles, which is considered the final implementation. This report outlines the importance of optimization by comparing the different methods and makes suggestions on how to improve the final implementation.

## 2. Practice

The project outcomes are a product of the research report findings, academic research and implementation, and supervisor feedback.

### 2.1. Project Outcomes

The outcome of this project is a fully working configurable fluid simulation system based on SPH. There are three versions of the SPH solver, the first one being plain SPH, the second one introducing spatial hashing and the third one using an imported C++ DLL which also uses spatial hashing. Each version can handle up to 300, 450 and 1000 particles, respectively (Figure 1). A scene that simulates a game environment has also been implemented in order to showcase the results. The user can configure certain values of the simulation to enable full control over the fluid behaviour. Options are available to change parameters like particle count, starting shape, viscosity, bounciness, etc. The simulation can be

placed anywhere within the game environment, featuring full compatibility with Unity's own collision system. This excludes the C++ implementation where the particle number and colliders must be pre-defined. The user also has full control over the behaviour of the simulation and can restart it at any time, even during gameplay, which is useful when making small parameter changes, in order to achieve desired results.

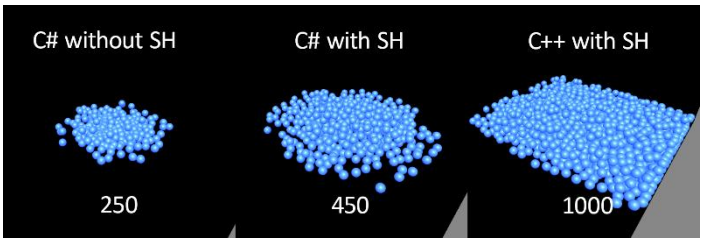



Figure 1. The 3 different implementations comparison. Top text – type; Bottom text – number of particles.

### 2.2. Research Report Outcomes

The research report was essential when obtaining knowledge to satisfy the start of the implementation. The research report outcomes established a strong understanding of fluid dynamics and can be summarised in the following three questions.

**Why SPH and how does it work?**

In the research report grid based, particle based, and hybrid methods were all considered as viable fluid simulation methods. The grid-based method follows a Eulerian approach which confines the fluid to fixed points on a grid and examines the fluid flow based on changing values over time (Bridson *et al*, 2007). The particle-based method SPH represents a fluid as a mass of particles that simulates fluid behaviour when each small particle inside the mass interacts with each other (Stam, J., 2003). The hybrid methods, which examine aspects of both grid and particle-based methods usually have discrepancies between accuracy and performance, therefore were initially discarded. Between grid and particle-based, the boundless SPH approach was chosen due to currently being the popular novel approach in industry as well as being accurate and optimal with higher visual fidelity and fairer computational speed compared to other methods.

The SPH method is based on the Navier-Stokes equations (Equation 1). It can be broken down into five core steps:

- Compute density
- Compute pressure
- Calculate velocity by combining density, pressure, and external forces (gravity)
- Check for collisions with objects and apply appropriate forces
- Apply the new estimated position to the particle in the game scene

Executing these steps for each particle per time step is the minimum requirement to be able to simulate a fluid using SPH. There are additional steps that can be added to optimise performance or enhance visuals.

$$\nabla \cdot \mathbf{u} = 0 \quad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \quad (2)$$

Equation 1. Basic Form of the Navier-Stokes Equations. (Stam, J., 2003)

**How to apply SPH to a game environment effectively and efficiently?**

Simulating a fluid in real-time is a very compartmentally heavy operation and doing so alongside other gameplay logic or systems can cause significant performance issues. To mitigate such issues, the simulation can only run with a limited number of particles to minimise performance loss during gameplay. Due to that number being relatively low, optimisation methods are implemented to increase the performance of the simulation, which allows more particles to be added alongside keeping the performance stable. The research report highlighted two possible methods that can be implemented to optimise the SPH solver. The methods are spatial hashing and the inclusion of an external plugin, which compiles the solver in C++, producing better performance. With the implementation of those two methods, the performance increases significantly. They are later compared against each other to evaluate why they were appropriate choices and how they can benefit performance.

**How does Unity's pipeline work and how does that affect the project's outcome?**

Unity is currently one of the most popular game engines used to create applications and games for its C# component-based system. It offers clear scripting and engine cross-integration, as well as cross-platform compatibility. It is also free to use; therefore, it was chosen for this project. In Unity, there are different systems that come "out of the box", such as the collision and physics systems. Unity's collision system is simple and easy to use, therefore widely used by Unity developers. The fluid simulation implemented uses custom collision system, but it is fully compatible with Unity's in-built one with the usage of object tags. Object tags can be placed on any object, therefore, anything marked with the "Collider" tag will become a collider object that can add force to the particles. It does not work with volumes, but current integration allows for any quad or plane object to act as a collider.

The gameplay scene is made entirely using the Unity renderer, so anyone can create or import their own custom scenes to use the fluid simulation in. The built-in Unity inspector allows the user to change parameters. The fluid has exposed parameters which the user can manipulate to influence how the fluid behaves (Figure 2). This will allow users to manually change the fluid behaviour at any time to best fit their use case. It also allowed simulations to be tested visually during development to iterate over the design and fix bugs.

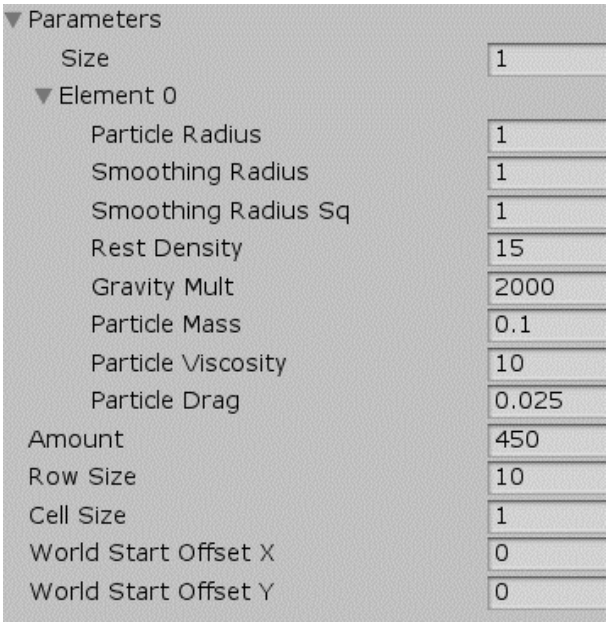



Figure 2.Configurable fluid parameters in the Unity Inspector.

### 2.3. Implementation

The implementation started with breaking down specific research and implementing research findings. One of the main sources used to break down SPH and understand how to implement it was a book on fluid dynamics by Kim (2016). The book was a good starting point to understand how SPH works, but with its very detailed implementation of SPH, it was also an inspiration when implementing the kernel functions and the neighbour search algorithm. This work utilizes the same concepts but builds the implementation from ground up.

### SPH Forces

The code implementation for forces applied to particles is located in the script's Update() function. The logic follows the pseudo code below:

```
Foreach Pi particle in particles do
 Set density, pressure to 0                          /*initialize*/
 Create game objects in scene
Foreach Pi in neighbours(Pi) do
   density(Pi) =  density + mass * kernel    /*accumulate density*/
   pressure(Pi) = k^gas * ((density_i /density_o)^γ − 1)    /*calculate pressure*/
 End
Foreach Pi in neighbours(Pi) do
   Calculate pressure force                          /*calculate combined force*/
   Calculate viscosity force
   Calculate gravity force
   Forces(Pi) = force pressure + force viscosity + force gravity
Foreach Pi in particles do
      Velocity(Pi) = combined force / density * delta time
      Position(Pi) = velocity * delta time    /*integrate velocity*/
    End
Foreach Pi particle in particles
       Foreach n collider in colliders do
        If (Pi intersects with n)                    /*collision*/
              Dampen velocity(Pi)
              Position(Pi) = new position
       End
End
Foreach Pi particle in particles /*apply position to GameObject*/
   Apply Pi position to game object(Pi)
End
```

Figure 3. Pseudo code for SPH.

The pseudo code (Figure 3) outlines the steps implemented to calculate forces for all particles. The calculations follow the SPH methods outlined by Kim (2016), Stam (2003) and this project's research report.

### Kernel

The current SPH implementation works through the five steps outlined previously. The particles affected by those steps are particles in the kernel radius. The kernel radius serves to create the 'smoothness' that spreads out attributes to particles stored nearby. Each particle in the radius is affected by others in the same radius, and it is the core function that creates the smooth distribution of values observed in SPH (Figure 4).

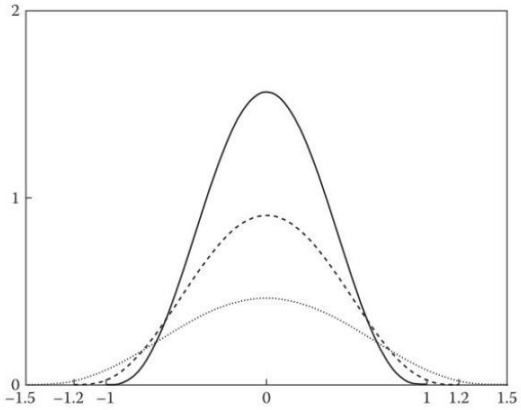



Figure 4. Kernel with different radii. 1.0, 1.2 and 1.5. As radius increases, the maximum value decreases. (Kim, D., 2016)

The general equation for calculating the kernel is as follows:

$$W_{std}(r) = \frac{315}{64\pi h^3} \begin{cases} (1 - \frac{r^2}{h^2})^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases}$$

Equation 2 Standard 3D SPH Kernel function (Kim, D., 2016).

The main rule is that the volume integral for any kernel function should satisfy:

$$\int W(r) = 1$$

As suggested in the book, this general-purpose function can be used for density, but a specialised version called 'spiky' kernel works better for pressure and viscosity

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

Equation 3 Spiky kernel (Kim, D., 2016).

During the implementation, the kernel functions presented in the book became difficult to debug because they were held within a custom struct. An article by Montes (2018) showcased an implementation of SPH kernel functions which did not rely on creating a specific struct to hold them, therefore the final implementation uses the kernels showcased in the medium article. This improved performance and allowed easier integration by removing the need to hold the data in a struct. With the addition of the spiky kernel, the viscosity attribute can now control the thickness of the fluid much more accurately than a generic kernel. By using a single source for the implementation, the kernels work better with each other to produce smoother flow of particles.

### 2.4. Optimisation

After the implementation of the SPH solver was completed to a satisfactory degree, the fluid body could host up to maximum of 300 particles at 60fps. The 60fps rule was set as a requirement to compare the performance using different optimisation methods. Thanks to research and feedback from the project supervisor, a decision was made to implement spatial hashing and a C++ plugin for the solver.

#### Spatial Hashing

Spatial partitioning systems such as grids, quad trees, space divisions, and others are widely used to improve performance of algorithms (Teschner, M. *et al*, 2003). There are different systems for use cases, however the one chosen for this project is spatial hashing due to its optimisation of neighbour search in 3D space (Ihmsen, M. *et al*, 2011 and Ihmsen, M. *et al*, 2014). Originally, the 300-particle implementation uses $O(N^2)$ search, meaning that if there are N particles, and every particle could potentially be a neighbour to every other particle, the search gets extremely expensive when N gets higher. With the data structure of spatial hashing however, the search becomes linear: $O(N)$. The particles that are affected are only neighbours of any given particle. The algorithm takes all particles and 'hashes' them to a one-dimensional grid by giving each particle position value a unique key, called hash ID (Figure 5).

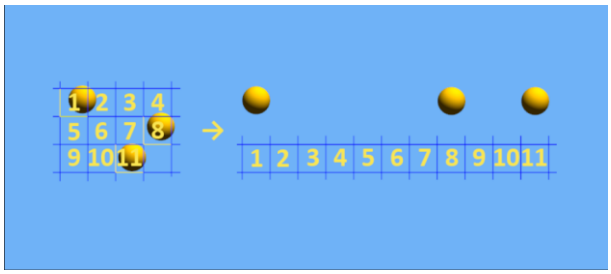



Figure 5. Hashing particles from 2D space to 1D (Sapiev, V., 2015).

It then uses the position of a particle to determine which particles are in the same range by their IDs and returns values for only the ones that are directly around the particle in 3D space, if there are any (Figure 6).

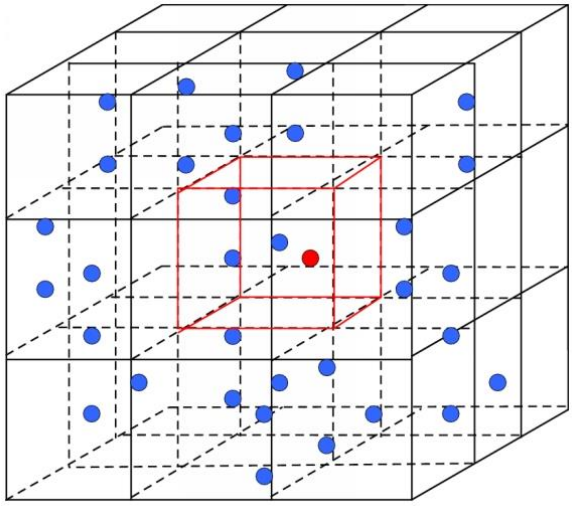


Figure 6. Neighbour search. Each cube represents a bin of particles. For the red particle, every other particle in the same bin is considered, as well as ones in the neighbouring bins (Lui, X. *et al*, 2015).

Different approaches were originally implemented and tested (Lefebvre, S. and Hoppe, H., 2006; MacDonald, T., 2009; Anon., 2009). All of them were either not functional or incomplete. An approach by Sapiev (2015) was later found to be a clear and more concise implementation of the spatial hashing method. However, it used the hashing algorithm and did not include implementation of the neighbour search

algorithm, therefore custom neighbour search functions were developed following the book by Kim, D. (2016). The results were successful, bringing the total number of particles that could be simulated to 450 and greatly reducing the amount of complexity of the simulation (Table 1).

| Method | Number of Particles | Frames per second | Time |
|---|---|---|---|
| No spatial hash | 300 | 60FPS | Real-time |
| No spatial hash | 450 | ~30FPS | Real-time |
| With Spatial Hash | 450 | 60FPS | Real-time |
| With Spatial Hash | 850 | ~30FPS | Real-time |

Table 1. Results comparison between simulations with and without spatial hashing.

**C++ Plugin**

An original concept suggested by the project supervisor and outlined in the research report was to implement a Unity compatible plugin that can improve the performance of the code by utilising a more optimal version of the solver in C++. For this purpose, the C# code was re-implemented in a C++ project and then converted into a Dynamic Link Library (DLL). A DLL is a standalone library of functions that can be imported into Unity as a plugin if it supports the same architecture. The process allows a custom script written in C# and compatible with Unity to extract the functions from the DLL and make use of them. Each of the five core steps of SPH outlined earlier were divided into separate functions which can then be called using that method. It works in a similar way to the C# implementation, except the functions are imported from an external DLL. This method drastically improved the performance of the simulation, allowing the use of two times the number of particles at 60FPS in real-time (Table 2). This would allow the simulation to run with a reasonable number of particles alongside other gameplay systems in a game environment without noticeable loss of performance.

| Method | Number of particles | Frames per second | Time |
|---|---|---|---|
| No spatial hash C# | 300 | 60FPS | Real-time |
| With spatial hash C# | 450 | 60FPS | Real-time |
| With spatial hash C# | 900 | ~30FPS | Real-time |
| With spatial hash C++ DLL | 1000 | 60FPS | Real-time |
| With spatial hash C++ DLL | 1500 | ~42FPS | Real-time |
| With spatial hash C++ DLL | 2000 | ~30FPS | Real-time |

Table 2. Comparison between C# and C++ implementations.

As all the calculations are done from an imported DLL, attributes such as number of particles, starting shape, viscosity, bounciness, etc. are no longer accessible by Unity. A solution to that was to create getter and setter functions to expose them so they can be modified by the Unity Inspector. The only downside to this approach is that the particles in C++ are stored in a statically allocated array which does not allow the number of particles to be changed without re-importing and re-compiling the DLL. This is an inconvenience if the user desires to change the number of particles in the fluid on the fly. The same applies to the particle colliders which are also static and need to be explicitly initialised in the DLL.

**3. Discussion of outcomes**

The SPH fluid solver successfully follows the concepts set out in the book by Kim, D. (2016). The book follows other notable examples from papers for its solver (Ihmsen, M., 2014 and Stam, J., 2003) and neighbour search (Teschner, M. *et al*, 2003) which were also utilised. The final implementation achieves a fully configurable and optimal fluid simulation system inside of Unity. Fluid parameters are exposed and can be modified at any time to allow the user to easily integrate it into any scene. It includes full Unity support with the addition of only a single script for each integration that can be attached to any empty object in the scene to serve as the fluid manager. To make it simpler, the current simulation can only interact with two-dimensional plane objects as it uses a custom collision function. If a more complex collision behaviours is needed, further development must see additional collision types implemented.

The current implementation exceeds performance of other similar recent SPH implementations by Stock (2019) and Montes (2018) when running single-threaded on the CPU (Table 3). Further tests can be observed in Appendix C.

| Author | Particle number | Frames per second | Time |
|---|---|---|---|
| This implementation C++ | 1000 | 60FPS | Real-time |
| Stock, K. (2019) | 220 | 60FPS | Real-time |
| Montes, L. (2018) | 250 | 60FPS | Real-time |

Table 3. Different SPH Implementation Comparison

This implementation can allow for users with little to no experience to achieve accurate fluid simulation results in their own game environment. The implementation can also be used as a substitute to its current best Unity non-SPH competitor Obi-Fluid (Virtual Method, 2020) which is available under a one-time payment license and can achieve similar results, although featuring more customization options of fluid placement. By drawing this comparison, it is apparent that further improvements can be made to develop a similar system that implements a dynamic fluid emitter instead of pre-set starting coordinates.

This implementation shows that by default Unity's C# code runs slower compared to a C++ version of the same when it comes to complex mathematical computations for SPH. This shows that using an uncommon approach to import a not out of the box compatible, lower level language, may have a positive impact on performance. On the other hand, since the DLL code is imported, some static values like the particle and collider count needs to be pre-defined which can cause inconvenience to the average user. The advantage of configuration over performance in the C# implementation is why it is important to understand the use case scenario before deciding on which one to use. A possible further improvement is to overcome this issue by implementing a different method of accessing static variables from a DLL, making more variables dynamic, or possibly taking a hybrid approach between C# and C++.

Another improvement that could be made is the implementation of fluid surface algorithm to better represent the fluid visually. One such algorithm is marching cubes (Wang, W., 2012). It scans the volume and creates a scalar field of points where the fluid surface exists (Figure 7). The points can then be connected to form polygon triangles and faces that can be rendered to show a more fluid-like surface.

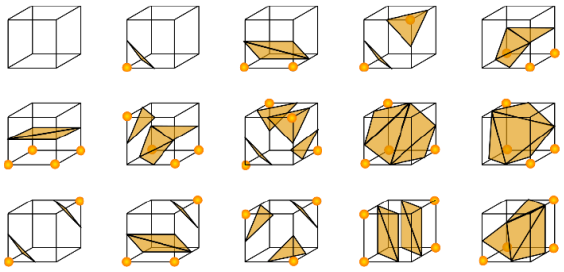


Figure 7. Marching cubes. The algorithm creates a polygonal isosurface by mapping points in a scalar field created using the fluid volume (Gourlay, M. 2012).

## 4. Conclusion and recommendations

The outcome of this project is a simulation that accurately represents SPH inside a game environment in an efficient manner. Thanks to the implemented optimisation methods, the simulation outperforms other current standard Unity SPH competitors when it comes to a single-threaded CPU implementation. This leads to the conclusion that the simulation can be used in a game environment. However, there are concerns that it may cause performance issues when used incorrectly, as the number of possible simulated particles drops when there is added complexity. For that reason, the final C++ DLL implementation is suggested to be used when a larger scale simulation is needed as it is the most optimal currently available solution to a single-threaded SPH implementation.

## 5. References

Anon., *My Internet Weblog*, 2009. Spatial Hashing in C++, Part 1. Available at: http://www.sgh1.net/posts/spatial-hashing-1.md.

Becker, M. and Teschner, M., 2007. Weakly compressible SPH for free surface flows. *Symposium on Computer Animation*, pp.1-8.

Bridson, R., Müller-Fischer, M. and Guendelman, E., 2007. Fluid simulation. *ACM SIGGRAPH courses*, pp.1-81.

Gourlay, M. (2012). *Fluid Simulation for Video Games*. [online] Intel Available at: https://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1.

Harris, M. (2007). *GPU Gems - Chapter 38. Fast Fluid Dynamics Simulation on the GPU*. [online] Nvidia. Available at: https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html.

Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A. and Teschner, M., 2014. SPH fluids in computer graphics.

Ihmsen, M., Akinci, N., Becker, M. and Teschner, M., 2011, March. A parallel SPH implementation on multi-core CPUs. In *Computer Graphics Forum* (Vol. 30, No. 1, pp. 99-112). Oxford, UK: Blackwell Publishing Ltd.

Kim, D. (2016), *Fluid engine development,* CRC Press, Boca Raton.

Lefebvre, S. and Hoppe, H., 2006. Perfect spatial hashing. *ACM Transactions on Graphics (TOG)*, *25*(3), pp.579-588.

Lucy, L.B., 1977. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, *82*, pp.1013-1024.

Liu, X., Wang, R., Li, Y. and Song, D., 2015. Deformation of Soft Tissue and Force Feedback Using the Smoothed Particle Hydrodynamics. *Computational and Mathematical Methods in Medicine*, 2015, pp.1-10.

MacDonald, T., 2009. *Spatial Hashing*. [online] GameDev.net. Available at: https://www.gamedev.net/tutorials/_/technical/game-programming/spatial-hashing-r2697/.

Monaghan, J. J. (2005). Smoothed particle hydrodynamics. *Reports on progress in physics*, *68*(8), 1703.

Monaghan, J.J. and Gingold, R.A., 1983. Shock simulation by the particle method SPH. *Journal of computational physics*, *52*(2), pp.374-389.

Montes, L., 2018. *How To Implement A Fluid Simulation On The CPU With Unity (ECS/Job System)*. [online] Medium. Available at: https://medium.com/@leomontes_60748/how-to-implement-a-fluid-simulation-on-the-cpu-with-unity-ecs-job-system-bf90a0f2724f

Onderik, J. and Durikovic, R.O.M.A.N., 2008. Efficient neighbor search for particle-based fluids. *Journal of the Applied Mathematics, Statistics and Informatics (JAMSI)*, *4*(1), pp.29-43.

Sapaiev, V., 2015. *Spatial Hashing*. [online] Unionassets.com. Available at: https://unionassets.com/blog/spatial-hashing-295.

Stam, J. (2003). Real-time fluid dynamics for games. *Proceedings of the game developer conference*

Stock, K. (2019). *Fluid Simulation / Smoothed Particle Hydrodynamics in Unity*. [video] Available at: https://www.youtube.com/watch?v=NJBz8rMJ0ZU

Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D. and Gross, M.H., 2003, November. Optimized spatial hashing for collision detection of deformable objects. In *Vmv* (Vol. 3, pp. 47-54).

Teschner, M., n.d., *Efficient data structures for SPH neighborhood search*.

Virtual Method, 2020. *Obi Fluid Physics Unity Asset Store*. [online] Assetstore.unity.com. Available at: https://assetstore.unity.com/packages/tools/physics/obi-fluid-63067.

Wang, W., Jiang, Z., Qiu, H. and Li, W., 2012. Real-time simulation of fluid scenes by smoothed particle hydrodynamics and marching cubes. *Mathematical Problems in Engineering*, *2012*.

**Bibliography**

Cheng, K. and Cairns, P.A., 2005, April. Behaviour, realism and immersion in games. In *CHI'05 extended abstracts on Human factors in computing systems* (pp. 1272-1275).

Grahn, A., 2008. Interactive simulation of contrast fluid using smoothed particle hydrodynamics. *UMEA University Sweden*.

**Appendix A: Project Log**

This project's activity log can be found online on the following website:
https://kristianbakov.myportfolio.com/creative-technologies-project-activity-log

**Appendix B: Project Timeline**

| | |
|---|---|
| 8th October 2019 | Outline proposal completed. |
| 14th October 2019 | Research into SPH implementations. |
| 24th November 2019 | C++ implementation following Kim (2016) completed. |
| 10th December 2019 | Research report completed. |
| 13th January 2020 | Core SPH implementation and demo video completed. |
| 15th January 2020 | Demo Showcase. |
| 10th February 2020 | Completed C# SPH implementation. |
| 12th February 2020 | Completed C++ SPH implementation. |
| 30th May 2020 | Completed spatial hashing implementation. |
| 3rd April 2020 | Completed C++ DLL with spatial hashing implementation. |
| 6th April 2020 | Game Environemnt Scene added. |
| 13th April 2020 | Performance tests and builds completed. |

**Appendix C: SPH Runtime Advanced Comparison Table**
All tests in this report are run on a single thread on the CPU. The simulations are run inside the Unity editor with identical scene conditions with no other scripts running in the background.
Hardware: CPU – Intel Core i7-8750H @ 2.20GHz; GPU – Nvidia GeForce GTX 1060; OS: Windows 10 Home; Display Resolution: 1920x1080p.

| Implementation | Number of particles | Frames per second | Time | Uses Spatial Hashing | Programming Language |
|---|---|---|---|---|---|
| Montes, L., 2018 | 250 | 60FPS (steady) | Real-time | No | Unity C# |
| Montes, L., 2018 | 300 | 45-60FPS (variable) | Real-time | No | Unity C# |
| Stock, K. 2019 | 220 | 60FPS (steady) | Real-time | No | Unity C# |
| Stock, K. 2019 | 300 | 40FPS (steady) | Real-time | No | Unity C# |
| Stock, K. 2019 | 900 | 5FPS (steady) | Real-time | No | Unity C# |
| This C++ implementation | 1000 | 60FPS (steady) | Real-time | Yes | C++ as DLL import & C# |
| This C++ implementation | 1500 | 42FPS (variable) | Real-time | Yes | C++ as DLL import & C# |
| This C++ implementation | 2000 | 30FPS (steady) | Real-time | Yes | C++ as DLL import & C# |
| This C# implementation (Spatial Hash) | 450 | 60FPS (steady) | Real-time | Yes | Unity C# |
| This C# implementation (Spatial Hash) | 550 | 40FPS (steady) | Real-time | Yes | Unity C# |
| This C# implementation (Spatial Hash) | 900 | 25-30FPS (variable) | Real-time | Yes | Unity C# |
| This C# implementation (No Spatial hash) | 300 | 60FPS (steady) | Real-time | No | Unity C# |
| This C# implementation (No Spatial hash) | 400 | 40FPS (varaible) | Real-time | No | Unity C# |

It is expected for fields marked with green to perform well in game environments, yellow might cause noticeable framerate issues, while red is not fitted for a gameplay environment. Those numbers may vary depending on external demand of the given scene. The tests were performed in scenes with minimal external graphical and computational demand.