Simulating fluids in Unity using SPH

Aron Bergman
arober@kth.se

Nils Tobias Forsberg
nilsfors@kth.se

March 2021

Project blog
Source code repository
Aiming for: A+ or bust

1 Summary

Generating a good simulation for fluid behaviour is an interesting and difficult endeavor of significant relevance to modern computer graphics. There are multiple paths one can take for abstracting the almost endlessly complex behaviour of a fluid into something computable. In this report, we discuss our implementation of one of those paths, called Smoothed Particle Hydrodynamics (SPH). Our implementation is primarily based on the method as described by Kelager[3] and Burak[2] The resulting simulation is considered a success, with the behaviour looking realistic and the computational optimizations working as intended. The simulation works in real time for a smaller number of particles, up to around 1000 depending on hardware, but requires precomputation to run smoothly at higher particle counts. Possible future work could include more optimization of parameter values, as well as GPU utilization.

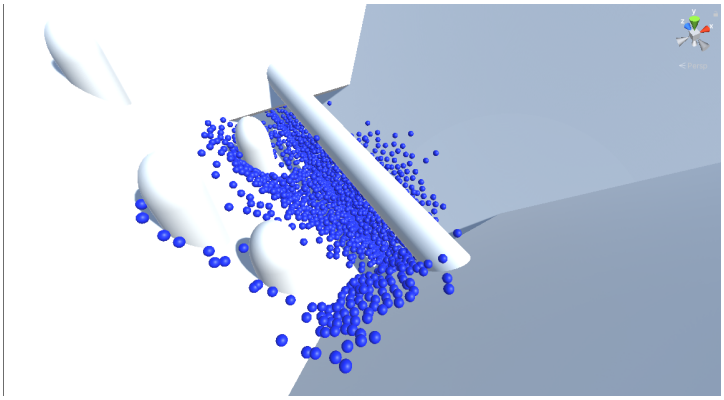


Figure 1: Water running down the left wall

2 Background & Theory

2.1 Fluid Simulation

Simulating fluids is a difficult endeavor. Compared to non-malleable solids, where the shape can be assumed to be the same across time and thus positional and rotational changes will be the same for the entire solid, the behaviour of a fluid is different throughout. Due to fluids being practically infinitely divisible, this means that describing a fluid perfectly requires an extraordinary amount of information. However, for modern games and visual effects, realistic simulations are important to get right, or the entire experience ends up looking cheap [2]. Our goal for this project is therefore is to create a fluid simulation that can provide realistic behaviour based on a modern fluid simulation approach.

2.2 Navier-Stokes

The Navier-Stokes equations are a pair of partial differential equations who put together describe the behaviour of a fluid. Of particular interest for us is the equation known as the momentum equation, which is given in Burak[2] for Newtonian fluids as

$$\rho(\frac{D\mathbf{v}}{Dt}) = -\nabla p + \rho g + \mu\nabla^2\mathbf{v}$$

This can be read as: The acceleration of the fluid at a point is the sum of acceleration from pressure forces (a high pressure resulting in a "spreading" force), gravity and viscosity forces (which acts as a fluid analogue of friction).

The other equation is much simpler, here as given by Kelager[3] for incompressible fluids:

$$\nabla \cdot \mathbf{u} = 0$$

In practice, this simply means that mass is conserved.

Since these equations are dependent on both time and all 3 space dimensions, solving these equations are quite complicated. As pointed out by Kelager[3], it is impossible to give a full solution to them for a fluid. However, for the purposes of simulation we are typically not interested in modelling a fluid with infinite precision and can instead use numeric approaches.

2.3 Approximating Navier-Stokes

There are multiple approaches to approximating the solutions to the Navier-Stokes equations, with Burak[2] describing the primary categories as being grid-based approaches and particle-based approaches. According to Kelager[3], while grid based approaches can have more accurate results in some aspects, they have several minutes of computation time per frame at high details, meaning it does not lend itself well to interactive behaviour. We have instead implemented one of the more popular particle-based approaches, known as Smoothed-Particle Hydrodynamics (SPH).

In SPH, the behaviour of the fluid is approximated via a discretization of the fluid to a finite number of positions in 3D space (here referred to as particles) and then approximating a solution to the Navier-Stokes equations at those positions only. It is a Lagrangian, mesh-free approach originally created to simulate astrophysics but has started to see a lot of use in fluid simulation as well[3].

2.4 Interpolation

As described by Kelager[3], SPH is essentially an interpolation method, meaning that values such as density, pressure, viscosity force etc. at each point are derived as an integral over the fluid space with further away points being weighted less than closer ones. In SPH, since we are using a discretized approach, an integral over the fluid is approximated by a summation over the particles which make up that fluid.

$$A_s(\mathbf{r}) = \sum_j A_j V_j W(\mathbf{r} - \mathbf{r_j}, h)$$

This equation can be read as: To approximate the value of an attribute A at a point $\mathbf{r}$ (represented by a particle), we sum the value of that attribute at all other particles j, multiplied by the volume which that particle represents, multiplied by something known as a smoothing kernel (more on this in the next section).

In practice however, the interpolation is a bit more complicated than this as we do not approximate the same value as we are summing. For example, in estimating the density at a point, we sum the mass of nearby particles rather than the density.

A complete mathematical description of SPH is beyond the scope of this course and involves a lot of multivariate calculus. Kelager[3] includes derivations of the interpolation equations for the constituent parts of the Navier-Stokes momentum equation. For our purposes, the end result is enough.

2.5 Kernels

As seen in the previous section, interpolation utilizes something known as a kernel (denoted W), which is a function that takes two parameters, the first being a vector from a particle j to the particle being evaluated and the latter being a number h which is the smoothing distance.

The main purpose of a smoothing kernel is to weight the impact a particle j should have when interpolating the values of another particle i, based on the distance between them. In simple words, when interpolating for example the pressure of a particle (which is a function of density), the density of other particles that are very close should be weighted more than those particles which are further away.

Smoothing kernels have two important properties:

$$\int_\Omega W(\mathbf{r}, h) = 1$$

and

$$W(\mathbf{r}, h) = 0 \quad ||\mathbf{r}|| > h$$

The first of these equations imply that the kernel should not (on average) amplify or diminish the end result of the interpolation, but rather weight the relative impact of particles on the interpolation based on their proximity. The latter states that for particles whose distance is greater than the smoothing distance, their contribution is always zero. This is important for computational purposes, as it means we do not have to care about all particles in a fluid when interpolating, only those which are within the smoothing distance.

3 Implementation

The implemented algorithm is based on the algorithm presented by Burak[2], where the density and pressure is calculated first, then the internal (pressure and viscosity) and external (gravity and surface tension) is calculated. Lastly, we perform the integration, collision detection and the XSPH velocity correction.

3.1 Mass, Density & Pressure

The second Navier-Stokes equation requires that mass be conserved across time. However, in our simulation this is easily done, as we can simply initially assign a mass to each particle and then not alter it. The mass given to each particle on initialization was decided on as:

$$m_{particle} = \frac{\rho_{rest} V_{fluid}}{n_{particles}}$$

where ρ_{rest} is the resting density of the fluid assigned via a parameter, V_{fluid} is the total volume of the fluid that the particles represent and $n_{particles}$ is the total number of particles. The reasoning behind this is that under the assumpion that total volume times average density must equal total mass, and so by dividing by the number of particles we can give the right number of mass for each particle to achieve that.

Density and pressure are not consistent across time or between particles, and thus has to be updated continuously in each timestep. Density is found via interpolation at each particle, according to the formula in Burak[2]:

$$\rho_i(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r_j}, h)$$

However, since the mass of each particle is always the same, a computationally faster method is to multiply by mass only at the end of the summation.

The pressure is simpler to calculate, here again as found in Burak:

$$p_i = k(\rho_i - \rho_0)$$

In other words, pressure is linearly dependent on the difference between the actual density and the resting density, with the coefficient k being a constant representing the unchanging parts of ideal gas law [3]. Had our simulation included for example temperature, this value might have been changing with time, but for our purposes it was simply tweaked through trial and error for good behaviour.

While the functions for density and pressure are relatively easy to implement, they proved to be some of the harder to achieve good behavior for. Since the two values appear in most other calculations, their impact is quite significant on the end result. One question was if the particle itself should be included somehow when calculating local density (otherwise, the density might turn out to be zero if a particle has no neighbors, despite the particle itself having a mass). The formulas suggested that it should not be included, but behaviour seemed better and more stable when it was, so ultimately it was included.

3.2 Forces

The main part of the simulation is the application of the forces that constitute the right side of the Navier-Stokes momentum equation in order to find the vector-valued acceleration for a given

particle. Its three constitutent parts were calculated individually. To implement this we used the equations as derived in Kelager[3], shown below:

$$\mathbf{f}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_{i \neq j} \frac{p_i + p_j}{2} \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h)$$

$$\mathbf{f}_i^{viscosity} = \mu \nabla^2 v(\mathbf{r}_i) = \mu \sum_{i \neq j} (\mathbf{v}_j - \mathbf{v}_i) \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h)$$

$$\mathbf{f}_i^{gravity} = \rho g$$

In addition to these 3 forces, a surface tension force was added to improve behavior. This was also implemented as recommended by Kelager[3]. The surface of the fluid is detected by a color field, where the the color field $c = 1$ where there is fluid, i.e. exactly at particle locations, and $c = 0$ everywhere else.

In SPH, the color field for particle i is calculated by

$$c_i = \sum_j c_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) = \sum_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h).$$

The gradient of the smoothed color field gives the inward surface normal

$$\mathbf{n}_i = \nabla c_i = \sum_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h).$$

Similarly, the laplacian of c_i is

$$\nabla^2 c_i = \sum_j \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h).$$

The surface tension force is then defined by

$$\mathbf{f}_i^{surface} = -\sigma \nabla^2 c_i \frac{\mathbf{n}_i}{\|\mathbf{n}_i\|}$$

where σ is the surface tension coefficient. When $\|\mathbf{n}_i\| \to 0$, which happens for particles surrounded by fluid, the fraction becomes numerically unstable. Kelager proposes that the force should only be calculated when $\|\mathbf{n}_i\| > \ell$, where $\ell > 0$ is the threshold.[3]

3.3 Scene

To test the particle simulation, we created three different scenes. The first scene contains a pool asset obtained from Unity's Asset Store. Using a pool felt natural, as we wanted to simulate water. Another scene with simpler geometry was added since the complexity of the asset was a confounding factor during testing. Lastly, a third scene was added late in the process to test more dynamic fluid behaviour and collision with varying objects.

3.4 Collisions

Collision detection is quite important for these simulations, since we want to confine the particles inside an object such as a glass, a box or, in our case, a pool. Collision detection is also quite hard to do, especially with complex objects. Since we wanted to focus on simulating the particles, we decided to use Unity's built-in collision detection.

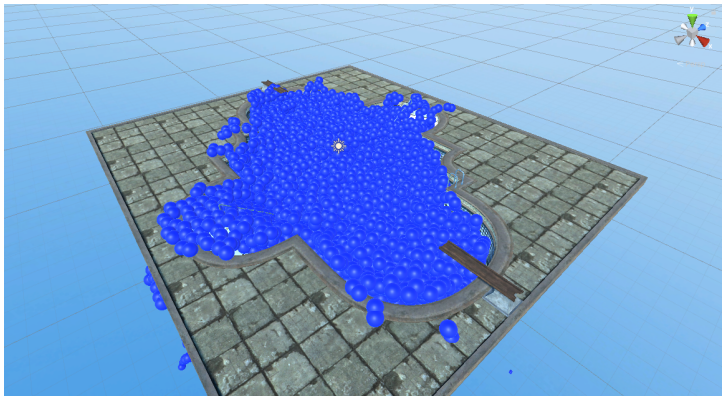


Figure 2: The pool scene

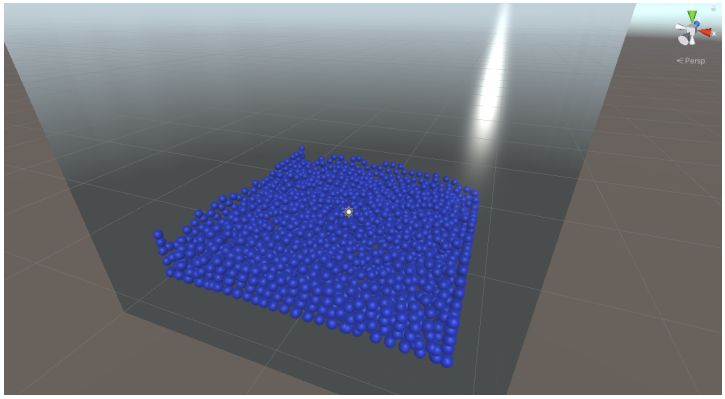


Figure 3: The box scene

The default behaviour of Unity isn't to only detect collisions, but also to handle them. The object holding the particles, such as the pool, had to have only a collider attached to it. In Unity, this is called a "Static Collider"[4]. In addition to a collider, the particles also had a rigid body attached to it. The default behaviour of the rigid body is to handle the collision, but since we want to do that ourselves we have to mark it as kinematic, which will disable automatic collision handling. In Unity, this is called a "Kinematic Rigidbody Collider"[4].

Note that Unity by default doesn't recognize collisions between "Static Colliders" and "Kinematic Rigidbody Colliders"[4], but this can be enabled in the project settings.

Unity fires one of three events when it detect a collision, `OnCollisionEnter`, `OnCollisionStay` or `OnCollisionExit`. Since we do our own collision handling and we don't want it to happen in the middle of a calculation, we have to store the collision so that we can process it later. So, when we receive one of the first two events, we store the collision data in the affected particle and access it later. When the third event is fired, the data is cleared from the particle, indicating that no collision has occurred.

We implemented three different ways to handle collisions. Aron wrote the first one, which used a spring force to motivate the particles to move away from the collision point.[1] We quickly threw

[1] Inspired by the boids in Lab 1.

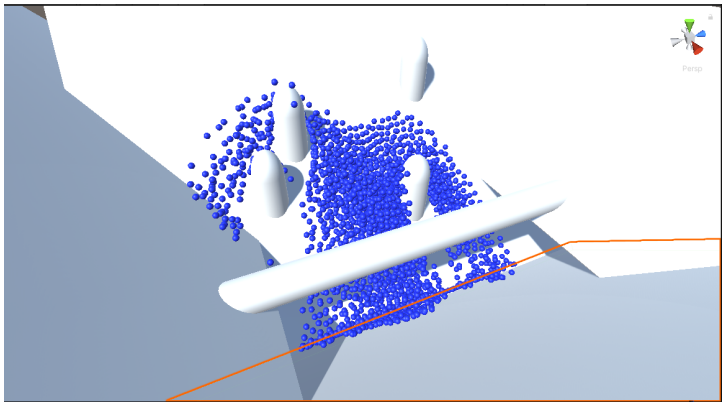


Figure 4: The scene with multiple obstacles

away that idea for two reasons. Firstly, the particles were able to move outside the bounding object, which resolved the collision, before moving back inside. Secondly, introducing forces, and thereby energy, due to collisions isn't a good idea from a simulation standpoint, because realistically the particles should loose energy upon collision.

So Tobias came up with a different strategy where the particle was moved along the collision normal to a position where it didn't intersect anymore. The velocity of the particle was then projected onto the plane that is perpendicular to the collision normal. The effect is that the particles spread out on the surface. An unfortunate side effect is that collisions between a particle and a vertical surface can make the particle move upwards on that surface, counteracting gravity.

It turned out that Kelager had defined a way to handle collisions. Like Tobias' collision handling, the first step was to move the particle along the collision normal to a position where it didn't intersect anymore. The next step was to reflect the velocity through the collision normal.[3]

The reflection is done with a normal vector reflection as a base

$$\mathbf{u}_i = \mathbf{u}_i - 2(\mathbf{u}_i \cdot \mathbf{n})\mathbf{n}$$

where $\mathbf{n}$ is the normalized collision normal. The problem with this equation is that it preserves the kinetic energy. We want a inelastic collision and to do that, Kelager proposes a coefficient of restitution, $0 \leq c_R \leq 1$.

$$\mathbf{u}_i = \mathbf{u}_i - (1 + c_R)(\mathbf{u}_i \cdot \mathbf{n})\mathbf{n}$$

When $c_R = 0$, the collision is inelastic, and when $c_R = 1$, the collision is elastic.[3]

It is still possible to introduce energy with this equation, since we reflect all of the velocity of the particle, but we only want to reflect the velocity that was generated during penetration of the object. Kelager has a solution to this, which we hasn't implemented.[3]

3.5 Kernels

The purpose of the smoothing kernel is to weigh the impact that two particles should have on each other during interpolation, which was implemented as functions with two parameters $\mathbf{d}$ and h, representing distance between the two particles and smoothing distance, respectively. Additionally, different Kernels are needed for different interpolations, and so a total of 5 kernel functions were created.

The kernels we implemented were those recommended in Burak[2], namely the poly6 kernel, its gradient and laplacian, the spiky kernel and the viscosity laplacian. The exact behaviour of

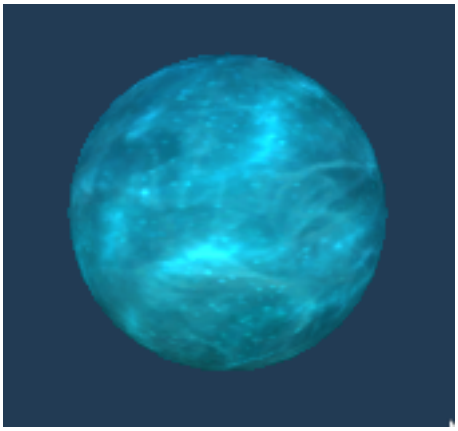


Figure 5: Initial water particle asset

these functions is rather obtuse and hard for us to decipher, with one online source claiming that they are being numerically shown to "just work" and not something one should look for too much intuitive understanding of [1]. For this reason, implementing them leaves room for very little else than directly translating them into code and trusting that they will work as intended. This was a relatively simple process, and worked without much effort. Since this set of kernels were deemed to work well enough, we did not attempt to implement any other ones.

3.6 Performance

Since the implementation is written in Unity, which is a real-time game engine, we started to see performance problems quite early. We wanted to simulate hundreds, hopefully even thousands, of particles, but doing the computations for each of these particles took too much time. Rendering that many particles also contributed to the performance impact.

To get a usable result from the simulation, we had to speed up the computations and the rendering.

3.6.1 Rendering

While the rendering didn't have the largest performance impact, it was the easiest to fix. Our initial particle was an asset made up of a intricate texture meant to look like water, see Figure 5. Simply replacing the texture with a solid color increased the performance of the simulation.

3.6.2 Parallelization

Tobias realized that many parts of SPH can be subject to parallelization, since many of the computations only depends on the current values of a single particle.

For example, the density and pressure computations can be parallelized, since they both only use the mass of their neighboring particles. Neither the mass or the neighboring particles will change during the computation of the density and pressure and can therefore be done in parallel.

The forces, on the other hand, have to be computed after the density and pressure of the individual particles has been calculated, since they sample the pressures and densities at the neighboring particles. Note that we still can parallelize the computation of the forces, but it has to be done after all computations of density and pressure has finished. Lastly, we can also

parallellize the integration stepping after the forces of all particles have been calculated. In aggregate, this results in 3 distinct parts of the primary loop – interpolating values for density and pressure, calculating forces on each particle and time stepping with those forces.

The effect of parallelization was quite substantial but wasn't enough to make it real-time, especially not for a large number of particles.

3.6.3 Spatial Hashing

One part of the computation that takes up a lot of time is searching for neighbors for each particle. The initial implementation for the neighbor search used the naive approach, namely that for each particle, we searched through all particles to find those within a certain radius. This means that the neighbor search had a time complexity of $O(n^2)$, where n is the number of particles. For large number of particles, this becomes quite slow.

Burak[2] addresses this problem with two different solutions.

The first one is a spatial hashing algorithm, where the world is divided into cells identified by a hash of their coordinates. This algorithm has a time complexity of $O(mn)$, where m is the average number of particles found and n is the number of particles[3].

The other solution is a hierarchical tree. The time complexity of this algorithm is $O(n \log(n))$. Since the time complexity for this alternative is worse than the spatial hashing, we chose to use the spatial hashing algorithm for our project.

The spatial hashing algorithm utilizes the fact that for every particle i, the smoothing radius h dictates that all particles at a distance $> h$ away from the particle i has no effect on it. By dividing the world into cells with a size of h in each direction, there is only a small number of cells that has to be searched to ensure that all possible neighbors are found.

As described by Kelager[3], there is two steps in the spatial hashing algorithm. The first step is to insert the particles by hashing and placing them in the correct cell. The second step is to query the appropriate cells for the neighbors of a specific particle.

Inserting To insert the particle we must first determine what cell it belongs to by discretizing its position using

$$\hat{\mathbf{r}}(\mathbf{r}) = (\lfloor \mathbf{r}_x/h \rfloor, \lfloor \mathbf{r}_y/h \rfloor, \lfloor \mathbf{r}_z/h \rfloor)$$

where $\mathbf{r}$ is the position of the particle.

The discretized position is then hashed by

$$\text{hash}(\hat{\mathbf{r}}(\mathbf{r})) = (\hat{\mathbf{r}}_x p_1 \text{ xor } \hat{\mathbf{r}}_y p_2 \text{ xor } \hat{\mathbf{r}}_z p_3) \bmod n_H$$

where p_1, p_2, p_3 are three large prime numbers and n_H is the size of the hash table used to store the cells.

The prime numbers used in this implementation are the same as those used by Kelager[3].

$$p_1 = 73\,856\,093$$
$$p_2 = 19\,349\,663$$
$$p_3 = 83\,492\,791$$

When the hash of the particle has been determined, we use it as an index into a hash table.

$$\text{table}[\text{hash}(\hat{\mathbf{r}}(\mathbf{r}))] = \text{Particle}_i$$

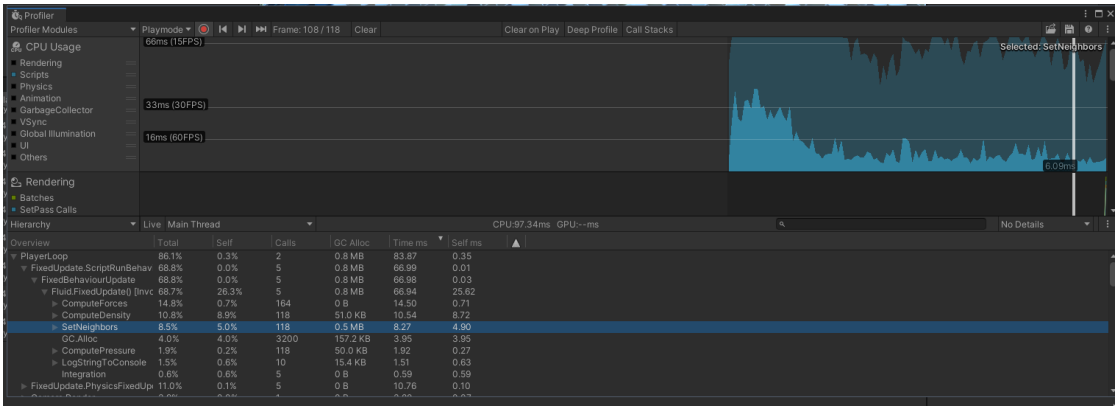


Figure 6: Performance profiling with naive neighbor search

Kelager[3] recommends that the size of the hash table, n_H, is a prime number larger than $2n$, where n is the number of particles. In an effort to try and increase performance, we decided not to do this. Instead, we measured the size of the space that the particles would occupy in the scenes and created the right amount of cells to exactly cover that volume. This reduced the number of cells that had to be iterated when updating the positions, but increased the likelyhood of "hash collisions", e.g. that particles that should belong in different cells share the same hash. The performance effect was noticeable enough that this seemed like a valuable tradeoff.

It should also be noted that the particle is *inserted* into the index, not *assigned*. As such, each index in the hash table needs to contain a data structure capable of holding multiple objects. We used a dictionary where each index contained a list, which in turn contained the particles inserted at that hash.

Querying Now that all particles have been inserted, we can query the spatial hash to find the neighbors of a particle. The bounding box for the particle is represented as two points,

$$BB_{min} = \hat{\mathbf{r}}\left(\mathbf{r} - (h, h, h)\right),$$

$$BB_{max} = \hat{\mathbf{r}}\left(\mathbf{r} + (h, h, h)\right).$$

Iterating from BB_{min} to BB_{max} generates all possible cells in which there can be particles that are neighbors to the particle with position $\mathbf{r}$. Finally, for each particle i in the cells of the bounding box, we check whether it is within the smoothing radius, $||\mathbf{r} - \mathbf{r_i}|| \leq h$. If that is true, we add it to the list of neighbors.

Performance profiling In Figure 6 we can see the result of Unity's performance profiling tool. We can see that the neighbor search takes about 8.5 % of the total time for each frame, or 8.27 ms. We can also see that the neighbor search doesn't have the largest performance impact, both computing the density and the forces take up more computation time. The main reason is that the profiling is done on only 250 particles. Since computing the density and forces is done in linear time, the neighbor search will become the worst performance offender when increasing the number of particles.

After Aron implemented the spatial hash and ran the performance profiling on 250 particles, the results in Figure 7 showed that the performance became even worse. The neighbor search takes up 22.1 % of the total time for each frame, or 73.14 ms.

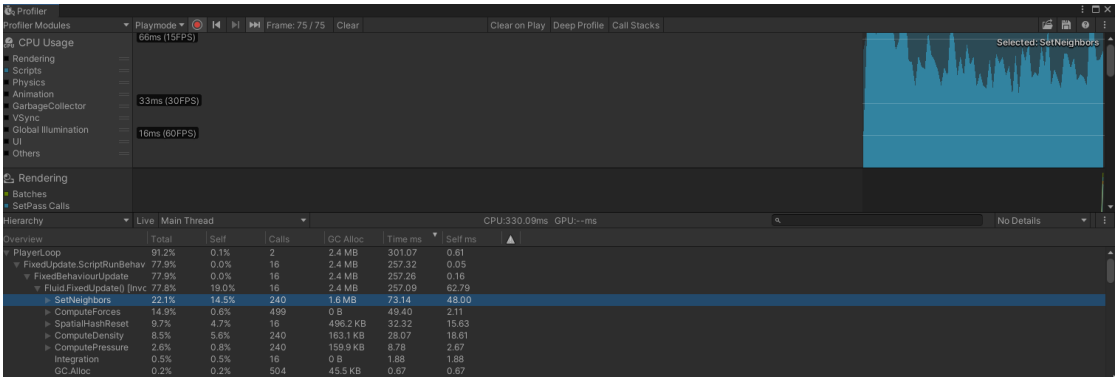


Figure 7: Performance profiling with spatial hash

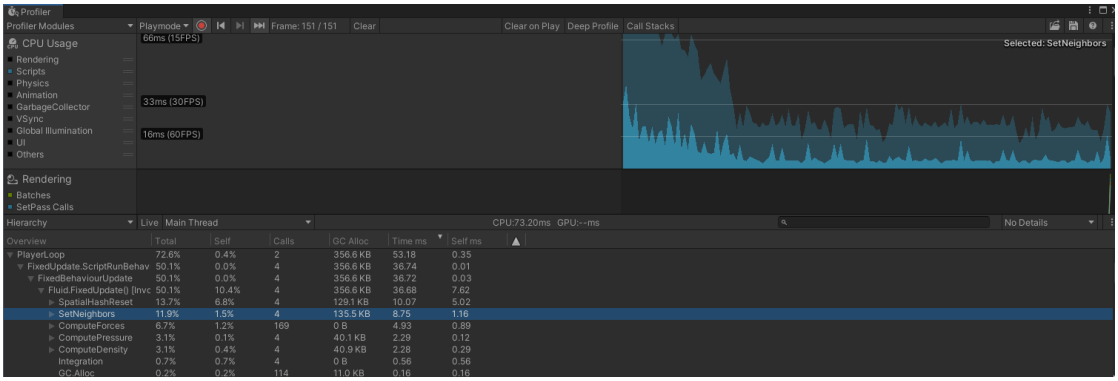


Figure 8: Performance profiling after adjusting h

Tobias suggested that it might be due to a too large cell size. Since the cell size is equal to the smoothing radius h, we had to decrease the smoothing radius. The smoothing radius was initially set to 3, which results in cells with a size larger than the bounding box for the generated particles, so all particles had all other particles as its neighbors. As suggested by Kelager[3], one approach for finding the smoothing radius is to solve for it as a function of roughly how many neighbors a particle should have. In practice however, this did not work great. The actual number of neighbors in the simulation was often wildly off from what the input would have suggested, and thus it was ultimately treated as a somewhat arbitrary parameter to be tuned for behavior and speed.

After shrinking the smoothing radius, and thereby the cell size, we were able to achieve much better performance, as seen in Figure 8. The neighbor search still takes up 11.9 % of each frame for 250 particles, but we can achieve much better performance with an increased number of particles than we could with the naive approach.

From Figure 8 we can see that there is one more thing in the spatial hashing algorithm that we can optimize further, namely the spatial hash reset.

The spatial hash reset is done at every simulation step to make sure that all particles are placed in the correct cell after their positions has been updated. Our initial method of resetting the spatial hash was to completely clear out the dictionary structure, repopulate it with empty lists and then insert all particles into their correct cells. Figure 9 shows the performance impact

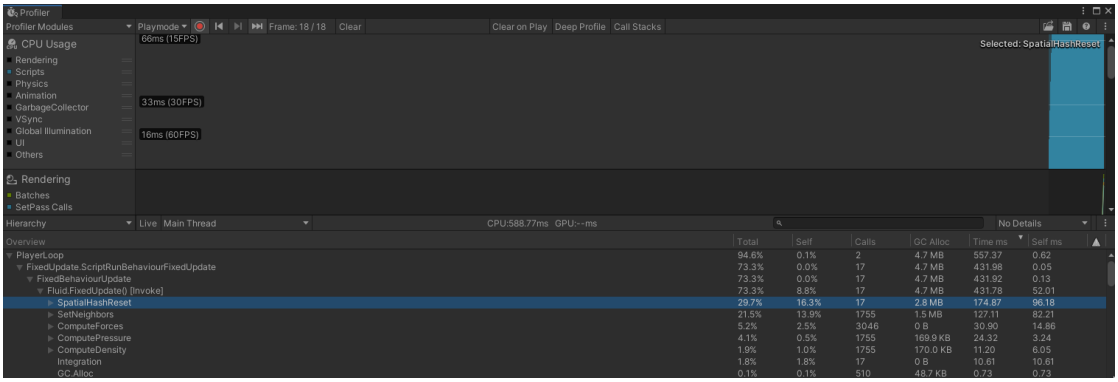


Figure 9: Performance profiling with initial spatial hash reset

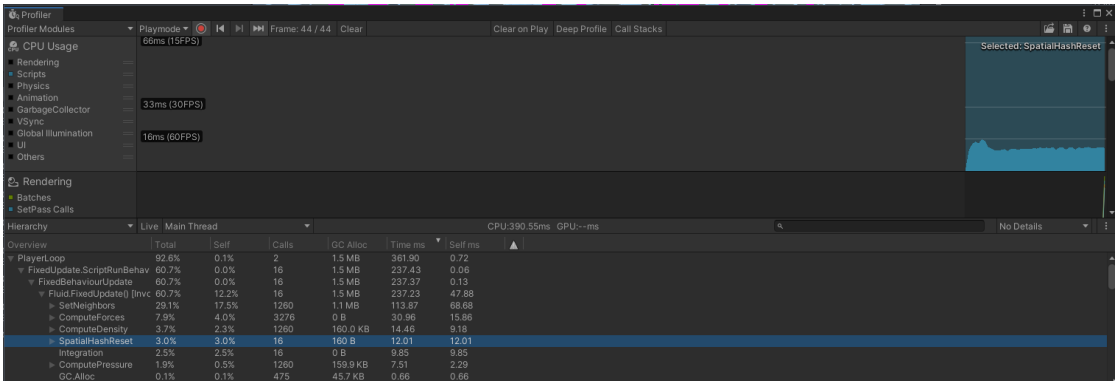


Figure 10: Performance profiling with efficient spatial hash reset

of this for 1500 particles, where 29.7 % of the time is spent on resetting the spatial hash.

Instead of clearing it out and the inserting everything again, we decided to iterate through all cells and move the particles that are in the wrong cell. The result of this change can be seen in Figure 10, where the reset only takes up 3.0 % of the time in each frame.

We also did some minor performance changes, like hard-coding the limits of the boundary box iterations and storing the boundary box for each cell. The effect of both changes were measurable but not significant.

3.6.4 Precomputation

When we felt that most of the code had been optimized and we had parallelized everything that we could, we still had performance issues. The frame rate was enough to see and understand the simulation but it didn't look good. From a simulation standpoint the result wouldn't be useful for anything, since most of the computed frames had to be thrown away by Unity to play the frames in real time.

The easiest solution would simply be to precompute all positions for a specified duration of time, store them and play them back when done. The only computations that had to happen during "game play" was to update the position from the stored values.

Our main concern when thinking about doing this was the fact that we were using Unity's

physics for collision detection and that it wouldn't be possible to invoke that whenever we wanted it to. As it turns out, Unity has the ability for programmers to invoke the physics module whenever they desire, so precomputing the simulation turned out to be quite easy.

The precomputation resulted in a huge performance boost. As expected, the start time increased drastically but when the simulation actually started it ran smooth. Unfortunately it didn't run as smooth as we had hoped for. The reason is that rendering a large number of particles, even with a simple shader, takes a lot of time. The resulting frame rate is just below 15 FPS.

A downside to precomputing the positions of the particles in the simulation is that it becomes impossible to interact with the fluid in real time.

3.7 Integration methods

Two integration methods were implemented, the forward Euler method and the Leapfrog method. The forward Euler method is the simplest possible method, and simply increments the position and velocity as:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta\mathbf{F}_t/\rho_t$$
$$\mathbf{r}_{t+1} = \mathbf{r}_t + \Delta\mathbf{v}_t$$

Where $\mathbf{v}$ is the velocity, F/ρ is the acceleration and $\mathbf{r}$ is the position.

The Leapfrog method is not too different, except that velocity and position are interleaved and incremented in every other time step. This was implemented as described in Burak[2]:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \frac{1}{2}(\mathbf{a}_t + \mathbf{a}_{t+1})\Delta t$$
$$\mathbf{r}_{t+1} = \mathbf{r}_t + \Delta\mathbf{v}_t + \frac{1}{2}\mathbf{a}_t\Delta t^2$$

Where a is the acceleration, calculated as F/ρ.

For the implementation in Unity, the time step was fixed to be 0.02s resulting in 50 updates per second. This is the update frequency of Unity's `FixedUpdate` function, which is recommended for physics simulations.[5] This means that the position values and velocity values were only incremented half as frequently for the Leapfrog method as for the forward Euler method. Burak[2] mentions that the forward Euler method is unstable and requires very small time steps. However, it seems that the time step we used was sufficiently small as the simulation appeared stable throughout even for this method. Ultimately, we decided to use the Leapfrog method instead, even though we struggled to notice any significant difference.

3.8 Velocty correction

As recommended by Burak, the XSPH velocity correction was added, which is performed on every particle prior to integration:

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \epsilon \sum_j \frac{2m_{particle}}{\rho_i + \rho_j}(\mathbf{v}_j - \mathbf{v}_i)W(\mathbf{r}_i - \mathbf{r}_j, h)$$

Epsilon is a value between 0 and 1 (in our implementation, 0.5), depending on how much you want the velocity to correct itself. This correction does not simulate any law of physics or such. Its goal is to produce a more stable particle flow, and in practice it worked quite well to produce that. The velocity correction has some similarities to the viscosity force, as when the two vectors v_j and v_i are similar the correction tends towards zero.

3.9 Constants

We wanted the simulated fluid to behave like water, so we have chosen the constants to fulfill that. Some of these values are based in reality, such as the density, while others aren't, like the gas constant. Many of the constants are taken from Kelager.[3]

Description	Symbol	Value
Density at rest	ρ_0	$997\,\mathrm{kg\,m^{-3}}$
Viscosity	μ	$0.8\,\mathrm{mPa\,s}$
Surface Tension Coefficient	σ	0.0728
Surface Tension Threshold	ℓ	7
Gas constant	k	30
Restitution coefficient	c_R	0

Table 1: Constants used to simulate water

4 Discussion

While the simulation can achieve fairly realistic behaviour, it depends on a somewhat arbitrary tuning of certain parameters values for the simulation to behave well. However, these values are not always "correct". For example, the density of the particles at rest should hover around the resting density, however in practice good behaviour can be achieved at much higher average densities than that. Quite frequently, a change would be made that should in theory make the simulation behave better, but which in practice made it worse. This makes for an interesting consideration – is the priority for the simulation to behave like reality, or to accurately simulate its parameters? In a perfect simulation, these two probably align. However, within our time frame for this project this was not always the case, and given the choice we tended to prioritize a good simulation. In the final version, the main negative consequence is that the actual volume is quite significantly smaller than what is suggested by the parameter, and density is generally a bit too high.

Initially, this project set out to simulate buoyancy of the fluid, on top of all the typical SPH functionality. However, reaching a satisfactory point for the SPH simulation took longer than expected, and so the scope was narrowed down to focusing on getting the SPH simulation to work well and to try to implement the various optimizations described in the implementation section. While it was a bit disappointing to have to narrow the scope of the project, we felt it more important to create a good baseline simulation than to try to add bells and whistles to something barely working.

As Kelager [3] mentions, utilizing the GPU could significantly speed up performance, although it would likely not be trivial to do so. It is possible that by pursuing this, a simulation with over a thousand particles might be able to run in real time.

Ultimately, Unity also proved to be as much of a detriment as a benefit. Using its built in collision detection and easy-to-build scenes made testing a bit smoother, and being able to tweak parameters via the inspector was welcome. However, many of our struggles during the development process surrounded making performance improvements, and within the Unity framework we were stuck using the relatively slow language C# rather than something more engine-suitable like C++ or Rust. Were we to attempt a similar project again, we would probably not have chosen Unity for this reason.

References

[1] Developer Central. *7 - Smoothed Particle Hydrodynamics*. URL: https://youtu.be/SQPCXzqH610?t=1465.

[2] Burak Ertekin. "Fluid Simulation using Smoothed Particle Hydrodynamics". MA thesis. Bournemouth University, 2015.

[3] Micky Kelager. "Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics". University of Copenhagen, 2006.

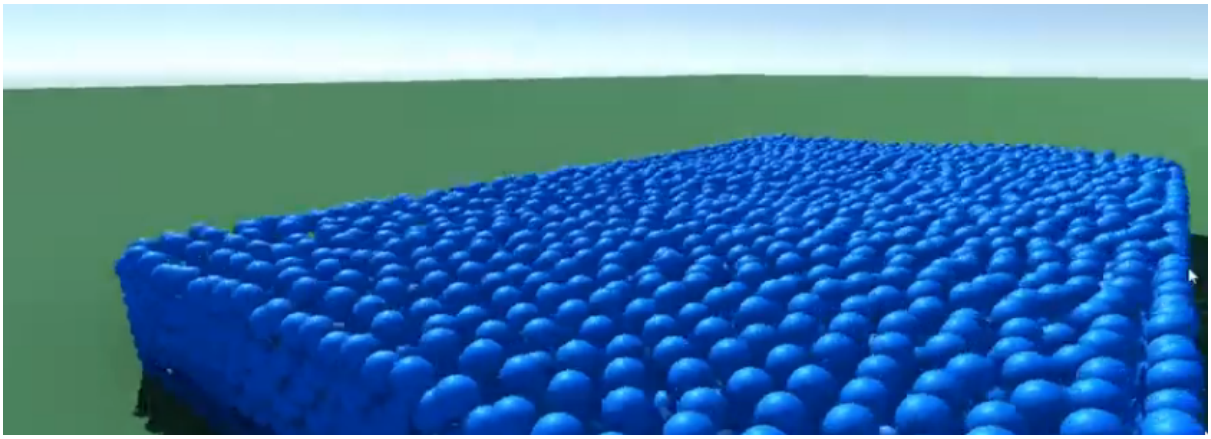
[4] Unity. *Colliders*. URL: https://docs.unity3d.com/Manual/CollidersOverview.html.

[5] Unity. *MonoBehaviour.FixedUpdate()*. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html.

Project specification

Sink or Swim in a Smooth-Particle Hydrodynamics simulation

Nils Tobias Forsberg (nilsfors@kth.se), Aron Bergman (arober@kth.se)



Background
Smooth-Particle Hydrodynamics (SPH) is a method used to simulate a continuous media using discrete particles. It is a mesh-free Lagrangian method, which means that we monitor how the particles move over time. It was originally used in astrophysics but has also been used to model fluid motion as well.

Problem
Under what conditions and parameters will rigid bodies of various sizes, densities etc sink or float in a SPH simulation.

Implementation
Our intention is to create the simulation in Unity. Our SPH model will be based on the paper by Burak Ertekin (2015) (https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc15/06Burak/BurakErtekinMScThesis.pdf), ideally experimenting with different integration methods to see which works best.

The first step will be to implement an SPH simulation for the fluid such that it can fill up a volume

The next step would be to throw ‘rocks’ into the water and watch it splash.

After this has been implemented, we will attempt to throw light objects onto the ‘liquid’, hopefully being able to simulate buoyancy in it.

Ideally, the force unto the body could be generated in much a similar way as between particles in the fluid, which would result in buoyancy emerging naturally. This would require the particles in the SPH model to accurately model pressure differences within the fluid however, which we are not confident they would.

Collision will have to be implemented between the fluid and the box which contains it. This will likely be done using the built-in Rigid Body functionality in Unity.
The pool will likely be constructed using a free Unity asset.

Tools
The project will be built in Unity 2019.4.21f1.

Risks and Challenges
The computational complexity of the simulation might get too large, resulting in the simulation not running well.

The process of applying buoyancy force might prove too difficult to do in a way that still behaves remotely realistic.

Translating the paper's quite dense language to code might take more time than expected.

Evaluation
The project will focus on evaluating how close the SPH implementation is to reality-based behavior. It is also interesting to evaluate the performance of the implementation.

References
B. Ertekin (2015) Fluid Simulation using Smoothed Particle Hydrodynamics Bournemouth University

T. Weaver, Z. Xiao (2016) Fluid Simulation by the Smoothed Particle Hydrodynamics Method: A Survey Bournemouth University

Project Blog
https://gits-15.sys.kth.se/pages/arober/modsim-blog/