Realtime Fluid Simulation on the GPU with *Unity3D*

Alex James

Cardiff University

Abstract The online simulation of fluid bodies is commonly used within games to create visual effects and puzzle mechanics. In this paper, we discuss and implement a method for achieving this system within the context of the popular *Unity3D* game engine, with a focus on creating a reliable and performative system capable of running alongside a typical game environment with an acceptable impact on frame times on consumer hardware.

1 Introduction

1.1 Motivation

Both the realistic and the stylized animation of fluids are oft sought-after components of such artistic media as animated films and video games. A plethora of approaches including those targeted at offline rendering and realtime simulation exist, and here we are attacking the latter with the common particle-based method for its less space-constricted nature and capacity for a high resolution result without unrealistic memory requirements. We stipulate in this paper that the result be of an acceptable performance on consumer hardware, as well as appearing believably realistic.
We also introduce a novel alternative approach to the neighbourhood generation that underpins the simulation, one that uses bitonic sorting to achieve memory contiguity of neighbourhoods rather than sorting algorithms in previous literature.

1.2 Background

One such method of simulating individual fluid particles is the highly cited Smoothed Particle Hydrodynamics (SPH), originally described by Monaghan [1992] [1]. It offers several advantages in its solution of conservation of energy, momentum, and mass while using a Lagrangian representation to allow for a dynamic domain of simulation. Such particle-based methods have numerous advantages over Eulerian approaches:

- Since each particle represents a fixed quantity of mass, conservation of mass is simplified.
- Properties are only stored at the positions of particles, rather than at every point on the grid.
- No computation is wasted on empty space

SPH does however leave some issues unresolved, particularly fluid surfaces exhibiting unrealistic behaviour due to neighbour deficiencies and fluctuations in pressure in part due to how the pressure is determined in terms of density from the stiff equation of state. Solutions to these include enforcing incompressibility via the solution of Poisson pressure equations and the weakly-compressible SPH (WCSPH) method wherein pressure is calculated directly from a stiff equation of state.

The dependency of WCSPH on stiff equations of state unfortunately results in forces of a size that limit the size of the time-step [Macklin & Müller 2013] [2], due to the Courant–Friedrichs–Lewy condition that limits the speed of the propagation of information through a body to the distance between the body's elements [Courant et al. 1967] [3] — information in a given spatial cell must propagate only to its immediate neighbours. This time-step limit thus increases the computational cost with decreasing compressibility, to the point that stiff, high-resolution fluids become impractically slow to simulate, especially for realtime applications.

A more recent approach, Predictive-corrective incompressible SPH (PCISPH), proposed by Solenthaler & Pajarola [2009] [4], propagates density estimations through the fluid body and updates then updates the pressure values such that the incompressibility constraint is satisfied. Notable advantages of the PCSIPH method include computation times that are orders of magnitude smaller than WCSPH, as well as the decoupling of the velocity and pressure field computations, all while avoiding compression-related artefacting.

In the PCISPH algorithm, positions and velocities of particles are projected forward in time and the future densities estimated. The deviation of this density estimate from the reference density is then calculated and used to update the pressure values of the particles and in turn the pressure forces. This process is applied iteratively until the density fluctuations are below a given threshold. The new velocity and position of each particle is then calculated for the following loop.

Another well researched and more cross-discipline problem is the determination of neighbourhoods within the particle simulation; to brute-force the interactions between particles has a time complexity of $O(N^2)$, and as particle counts increase the need quickly arises to find a means of limiting attempted interactions to only those neighbours within the smoothing radius of each particle.

Franklin [2005] [5] introduced the NEARPT3 algorithm, which utilises a uniform grid to organise points and then performs a fast query into any cells within a rough neighbourhood based on the total distance along each axis, then sorts those results by Euclidean distance. This sorted list is then iterated over and stop cells are determined by finding the last cell s_c whose closest point to the origin is at least as close as the farthest point of the current cell c. This requires only one pass and provides information as to where any points closer than c might be by stopping at s_c in the search.

Simon Green [2010] [6] targeted particle simulation specifically with his paper on using a parallel radix sort, implemented in *CUDA*. Again using a uniform grid to subdivide the simulation space into a grid of uniform cells, explicitly setting the size of each cell to be the same size as a particle - double the size of its radius. This "loose" grid allows for several assumptions to be made: Firstly, a single particle will only cover 8 cells (2x2x2 in three dimensions), and secondly that if overlap between particles is prevented via repulsion then it can be assumed that there will be a fixed number of particles per cell. These properties are useful in that they allow the use of relatively small fixed-size data structures to represent the grid, a feature desired when working with hardware acceleration. Additionally, we can devise that to search for neighbouring particles, we will have to search only the neighbouring 27 cells (3x3x3 in three dimensions). Using the grid indices of the particles, we can bin them into grid cells and easily determine the neighbours of a particle by checking the surrounding 27 cells. Green also refers to an alternative method of binning whereby particles are stored in every cell they occupy which reduces the computational cost of colliding the particles but increases the cost of building the grid, and estimates that this result is explicitly poorer when considering a GPU implementation. It is also stated that the grid itself can be incrementally updated rather than built from scratch at each update, which presents an opportunity for a performance increase, though rebuilding the grid ensures that performance is decoupled from the movement of the particles.

Building the grid in the first place is an additional hurdle, with strategies divided into two chief categories: with atomic operations, and via sorting. Using atomic operations, multiple compute shader threads build up two data structures: one that tracks the number of particles in each cell so far, and another that stores the particle indices at each cell, the latter of which takes advantage of the assertion that there can only be so many particles per cell and thus may use a fixed-size data structure. Particle indices are stored using scattered global writes, and the cells' particle counts are clamped to the imposed maximum. This approach has some performance implications, particularly in that the global memory writes are more or less random, and thus will not be memory-coalescent due to the probable spread of the data across memory and the

increased memory block requests that this brings. Green also notes that if multiple particles write to the same cell simultaneously, the writes will be serialized and bring about further performance issues. It is also proposed that a variable number of particles per cell could be supported by separating this process into two parts, using the atomic counting stage to inform a parallel prefix sum operation to ascertain the destination addresses for each particle, finally examining the particles and writing them to contiguous locations in the grid data structure using the previous step's results. While atomic operations are possible in *Unity3D* with compute shaders, this is platform dependent as Apple's *Metal* API does not presently support them in this context.

Alternatively, the grid can be built without any atomic operations by employing sorting; a solution with several passes, it begins by calculating a hash value from each particle's cell ID, or from the ID of the cell in a Z-order curve for more contiguous memory access. These hashes are then stored in global memory as a pair of unsigned integers consisting of the hash and the particle index. The particles are then sorted on their hashes, using a fast radix sort to create a list of particle indices in order. This list can be used to generate metadata regarding the start and end indices of individual cells, by dividing the processing into a thread per particle and compares the cell index of each particle with the previous particle in the list, determining if a boundary between cells lies there in memory simply by observing a difference between the cell indices and, if different, writing that cell index to another array in a scattered write operation. In a similar manner, the ends of each block can be found or alternatively computed by pairing the already-known starts of each block.

To improve memory efficiency in the rest of the interaction processing, the properties of the particles can also be sorted in the same order. This sorted list is then used when each particle checks the surrounding 27 cells for other particles to interact with. This method is noted for its high performance due to improving memory access coherence and reducing warp divergence as particles within the same warp have a tendency to be close together, spatially, and thus have a similar number of neighbours. Conversely, there are performance concerns regarding the stage where positions and velocities are looked up, due to the randomness of the accesses; Green proposes a fix to this wherein the arrays are bound to textures and retrieved using the *tex1Dfetch* function which improves performance hugely due to the caching of texture reads.

Hoetzlein [2014] [7]proposes replacing the radix sort with a counting sort; each particle is inserted into a cell and then the cell indices are counted to determine the number of particles in each cell. A prefix sum is then performed on these counts and a typical counting sort performed afterwards. The counting sort is of time complexity $O(n+k)$ where k is the maximum key value, meaning that the size of the grid cells must be chosen with care, potentially determined by testing on the frame times when implemented — Hoetzlein presents a graphic of the relationship between cell sizes and particle counts, showing the balance between particles per cell and grid cell checks per particle. In comparison to the parallel radix sort method's 15 kernel calls each frame, the counting sort method combined with atomic add operations for bin counts and indices performs only four kernel calls per frame. According to Hoetzlein's

benchmark, the counting sort method is 5-10 times faster than the radix sort method, able to process 400 million points per second.

To visualise the simulated fluid, a number of existing strategies exist that can be placed in one of two categories: those that generate traditional geometry, and those that use screen-space image effects; both such types have their applications and produce their own set of artefacts.
The marching cubes method, introduced by Lorensen and Cline [1987] [8], involves creating a polygonal mesh of an isosurface, which can be generated from a 3D scalar field of intensities. This can be used to visualise a fluid, but lends itself to Eulerian simulations due to its dependence on an explicitly discrete field, which is not inherently present in position-based particle simulation. Even if an infinitely granular Lagrangian particle system were to be discretised so as to map to the marching cubes method, the result would still be visibly coarse as the geometry will be of a resolution determined by the grid used to generate it.

A method that hybridises the screen-space and geometric methods is the screen space mesh technique by Müller [et al., 2007] [9], which intends to create a 2D triangle mesh using a technique derived from the marching cubes method, and then transform that mesh into world-space to apply shading and lighting effects. This approach was created for applications in games and other online-rendered media, and so has the merit of a view-dependent level-of-detail — an attractive feature to us, as our target application is the same and performance is a focus.

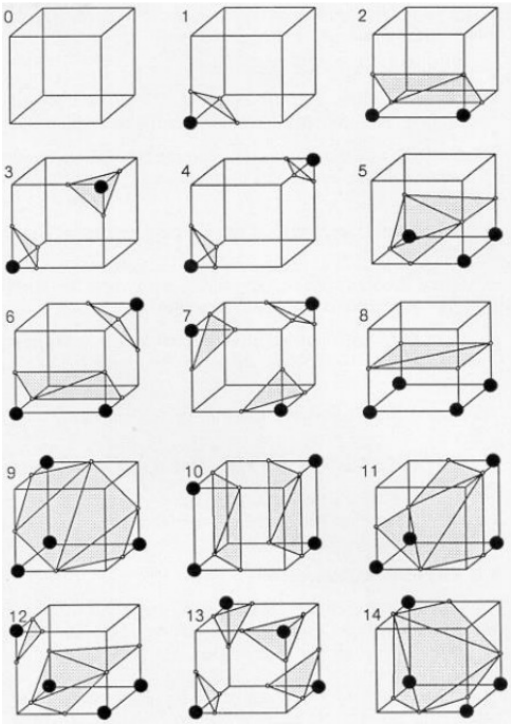

Figure 1: *Triangulation of cubes in the marching cubes algorithm. Taken from [Lorensen & Cline, 1987]* [8]

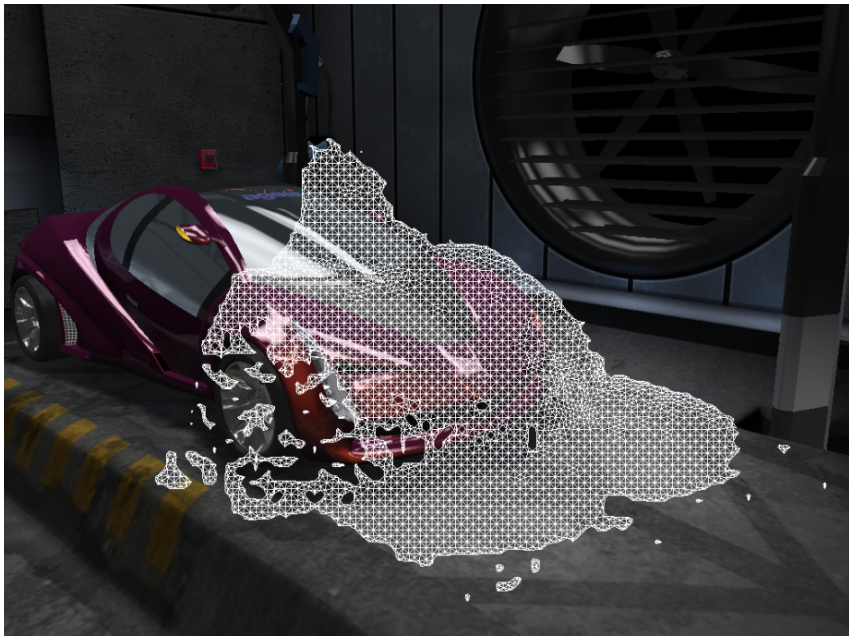

Figure 2: *A screen-space mesh. Taken from [Müller et al., 2007]* [9]

Other screen-space approaches include the sphere-imposter method used by Green [2010] [6], in which particle points are drawn to a buffer as sphere-imposters — a method of drawing spheres without geometry, such that the sphere is exactly as resolute as the buffer or texture it is being drawn to, and as such has an inherent distance-based level of detail. This method also only considers the surface closest to the camera, ensuring all processing is spent on visible, relevant results. Once the sphere imposters have been determined at each particle, their depths are then written to a buffer and, in a subsequent pass, smoothed to create a flatter, more convincing depth map. This depth map is then used to reconstruct the surface normals and positions, which allow for the standard Lambertian and Phong shading models, or a more modern physically-based rendering and image-based lighting approach, to be applied.
Much literature exists on such methods, and the primary part of their solutions that distinguishes them is their approach to smoothing the depth data. Van der Laan [et al., 2009] [10] propose an alternative to the common basis of weighting a Gaussian blur: curvature flow. This approach involves solving derivatives of depth across the map and gradually adjusting the depth so as to evolve along its curvature toward a flatter, more parallel surface. Other authors [Truong & Yuksel, 2018] [11] suggest pursuing the

Gaussian blur-based approach, using a narrow-depth-range filter to weight the depth samples used in the blur to retain the edges and discontinuities in the depth map while smoothing the lesser changes.

2 Approach

2.1 Eulerian and Lagrangian Fluids

The fluid behaviour we concern ourselves here is governed by the Navier-Stokes equations for incompressible fluids, which describe how the properties of the fluid's particles or quanta must change to match the materials constraints, such as resting density. To discretise these behaviours into something computable, we are presented with two established approaches: the grid-based Eulerian, and the particle-based Lagrangian. The former divides the simulation space into a grid, typically with fixed-sized cells, and iteratively solves each cell's state to enforce the various conditions. Conversely, particle-based solutions represent individual fluid particles, or more accurately clusters of molecules, each having a state that is solved in relation to its neighbours.

Both approaches have their merits and within the context of game development, but we will use a particle-based method here for several reasons: firstly, we are less interested in the mechanics and details of the volume of the fluid and more concerned with the surface and its æsthetics – we need the spatial granularity and resolution that a freeform particle approach provides, which a grid-based system would struggle with given its fixed cell positions. To achieve a similarly resolute result with a grid fluid would require extremely dense cells, perhaps exceeding the memory limits of the consumer GPU.

Additionally, the visualisation of the simulated fluid varies between the two approaches, as one offers discrete positions while the other simply describes the properties at each cell. Grid-based solutions can also easily map to a visualisation using marching cubes, since both are describing properties at fixed-space intervals, whereas particles lend themselves to less rigid methods such as screen-space effects, which we will use here for their ability to transform the output of the simulation into something æsthetically independent with a configurable and granular amount of smoothing, meaning the apparent quality of the surface is not reduced with a closer point of observation as a mesh-based approach would.

2.2 Particle Property Constraints

To begin the discretisation of the continuous field that is a fluid, we represent each particle as a collection of properties: a position r, velocity v, pressure p, and density ρ. For a fluid with a homogenous temperature, we can model the conservation of mass using the following identity:

$$\frac{\partial \rho}{\partial t} + \Delta \cdot (\rho v) = 0$$

Since we are representing the fluid as a finite number of particles rather than a grid, however, this conservation is guaranteed and need not be actively enforced.

We also model the conservation of momentum, this time using a form of the Navier-Stokes equation simplified for weakly compressible fluids:

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \Delta v \right) = -\nabla p + \rho f + \mu \nabla^2 v$$

Where *f* is external forces (e.g. gravity) and μ a constant representing the viscosity of the fluid. This can be simplified, again due to the use of particles as representation of the fluid; we may omit the convective term $v \cdot \nabla v$ and replace the partial derivative with the Lagrangian derivative of the velocity with respect to time:

$$\rho \left(\frac{Dv}{Dt} \right) = -\nabla p + \rho f + \mu \nabla^2 v$$

The right-hand side of this equation models the sum of the force density fields, consisting of $-\nabla p$ the pressure, external forces ρf, and the viscosity $\mu \nabla^2 v$. By applying Newton's second law, the acceleration of a particle *k* is therefore modelled as:

$$a_k = \frac{dv_k}{dt} = \frac{f_k}{\rho_k}$$

Here we assume some fixed properties for the particles, including a uniform size, smoothing radius, and resting density. For considering multiple phases or materials of fluid, this would have to be per-particle. These properties are operated on by performing a weighted sum of influences from all particles; the weighting we use is determined by inputting the distance between any two particles into a smoothing kernel. The properties at each position in space are determined thus;

$$S(r) = \sum_j m \frac{S_j}{\rho_j} W(r - r_j, H)$$

Where *r* is the position of the subject, *j* covers the range of indices of all other particles, ρ_j the density and r_j the position of the other particle and S_j represents a property at *j*'s position. The mass, m_j is constant throughout all particles in the simulation in our case.

As the density will vary with each simulation step, we enforce incompressibility by substituting density in as the quantity *S*;

$$S_\rho(r) = \sum_j m\frac{\rho_j}{\rho_j}W(r - r_j, H)$$
$$S_\rho(r) = \sum_j mW(r - r_j, H)$$

2.3 Interpolation

A smoothing kernel is used to attenuate the interaction between particles, which can be defined on a per-particle basis or as a uniform property of the system, depending on the implementation. We represent this smoothing radius as h and the weighting function for the kernel as W, and thus determine the quantity of a particle by summing their neighbours properties.
We will use the *Poly6* and *Spiky* kernels [*Matthias Müller et al., 2003*] [12] for density estimation and gradient calculation respectively, which are defined as follows:

Where r is the covector $\overrightarrow{p_i p_j}$, and H is the smoothing length.

$$Poly6(r, H) = \frac{315}{64 * \pi * H^9}$$

$$Poly6Kernel(r, H) = Poly6(r, H) * (H^2 - \|r\|)^3 \text{ for } 0 < \|r\| \leq H, \text{ 0 otherwise.}$$

$$SpikyGradient(r, H) = r * (H - \|r\|)^2 * \frac{-45}{\pi * H^6}$$

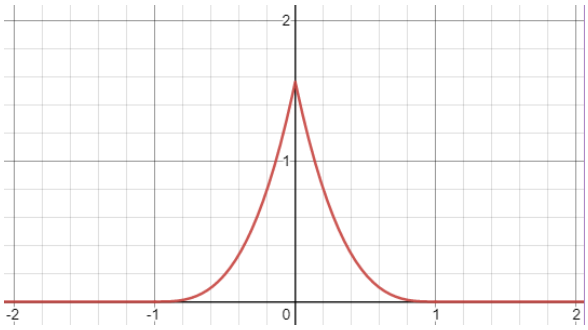

Figure 3: *A graph of the Poly6 kernel,where the horizontal axis is* ***r*** *on a given axis and the vertical axis is the output of the function.*

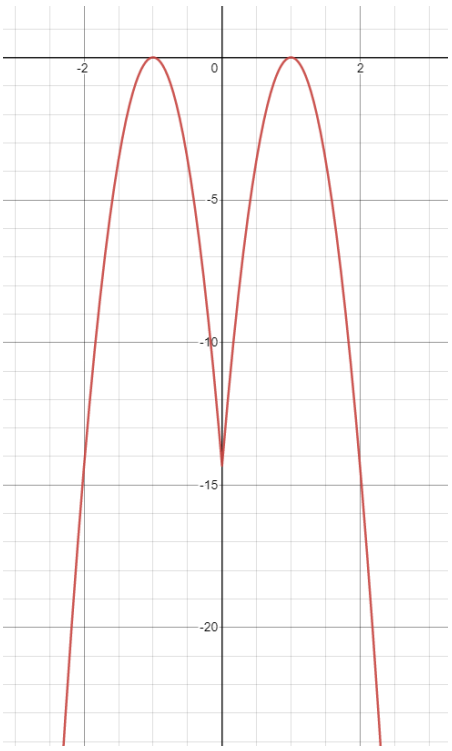

Figure 4: *A graph of the Spiky kernel, where the horizontal axis is* ***r*** *on a given axis and the vertical axis is the scalar value that would be applied to the vector* ***r***

2.4 Resolving Pressure

Substituting the pressure term into the SPH equation gives the following equation, that describes the quantity of the term $-\nabla p$, here as f_i^p:

$$f_i^p = -\sum_j m\frac{p_j}{\rho_j}\nabla W(r_i - r_j, H)$$

As noted in most literature, there is an inherent issue with the symmetry of this pressure calculation, as the smoothing function *W* yields zero when provided a distance of zero, meaning that a particle will only use the pressures of other particles when determining this term. If any two adjacent particles have a different pressure, then the forces applied to either particle will also be different.
Müller [et al. 2003] [12] proposes a simple solution that retains speed while ensuring stability in the simulation, the arithmetic mean of the two particles' pressures:

$$f_i^p = -\sum jm\frac{p_i + p_j}{2 * \rho_j}\nabla W(r_i - r_j, H)$$

When this pressure calculation is applied directly to a simulation, however, an undesirable expansion of the fluid is created. This has been previously solved in astrophysical applications with the application of a gravity force, maintaining something of an equilibrium within the fluid. As noted by Desbrun and Gascuel [1996][13], a material with a constant resting density would exhibit properties of the Lennard-Jones substance, i.e. a cohesive force, and thus they suggest substituting the usual $p = k\rho$ ideal gas state equation with $p = k(\rho - \rho_0)$, where ρ_0 refers to the resting density, which serves to not only naturally evenly distribute the particles within a body, but attempt to maintain the volume of the system after deforming forces are applied.

2.5 Viscosity

To ensure coherent motion and somewhat alleviate nonphysical oscillations in the simulation, we apply XSPH viscosity as defined by Schechter and Bridson [2012] [14]:

$$v_i^{new} = v_i + c\sum_j (v_i - v_j) \cdot W(p_i - p_j, H)$$

Where *c* is some small value — 0.01 in our case. This simply modifies the velocity before the next step by adding an aggregate of all surrounding particles' velocities. This new velocity must be treated as a separate variable until this step has completed for all particles, to ensure all velocities sampled are from the same time slice.

2.6 Visualisation

To visualise our position-based fluid, we use an extension of the screen-space method proposed by NVidia's Simon Green in his Game Developer Conference Talk [2010] [6], due to how it lends itself to parallelisation and the application of video games.
We begin by using geometry shaders to create billboarded (camera-facing) quads at the positions of the particles, on which we impose sphere impostors that are output to the game engine's depth buffer for sorting and to a separate depth map that we manually maintain. To this same buffer on a separate channel, we draw the sphere impostors again using additive transparency to accumulate a screen-space thickness of the fluid, for use later in shading. The depth map is then iteratively smoothed using the curvature flow method and the world-space normals at each pixel in the map computed; with this smoothed normal data, we translate the current scene's lighting and shadows into the same space as the map and shade the surface using Lambertian diffuse and phong shading in a full-screen image effect shader, and combine this result with thickness-based scattering and colour extinction to achieve the appearance of non-clear water or other coloured fluids.

3 Implementation

3.1 Initial Prototyping

We begin with a simplified simulation, in two dimensions and on the CPU, allowing us to prove the concepts of SPH while keeping the output relatively easy to debug and analyse. The first of the behaviours to implement is the neighbourhood generation and grid building, which we will achieve in the simplified model using the previously described counting sort method.

```
class Particle {
    public Vector2 position, newPosition, velocity, deltaPos, forces;
    public float lambda;
    public Particle(Vector2 position) {
        this.position = position;
        newPosition = new Vector2(position.x, position.y);
        velocity = Vector2.zero;
        lambda = 0;
        deltaPos = Vector2.zero;
        forces = Vector2.zero;
    }
}
```

Our particles are represented as a *Particle* object, with members representing the field quantities of the simulation. Once we've created the particles and given them initial positions at offsets from one another as well as defined the constants that govern the simulation, we can begin to use *Unity*'s *FixedUpdate* function to step the simulation at a fixed interval. To step the simulation, we first need to construct the neighbourhoods of the particles, which we will do by a much simpler and less efficient method than the counting or radix sort methods that simply involves checking neighbouring grid cells manually rather than relying on sorting.

We take several parameters that allow for the changing of the dimensions as well as the sizes of the cells, and we store the indices of the particles in lists representing each cell which in turn are held in another list representing the grid itself. This nested list approach is neither performant nor translatable to compute shaders, and will be replaced with a fixed data structure.

The function *GetCell* will allow us to query the 1-dimensional list of cells with two parameters describing the *x* and *y* indices of the cell we wish to retrieve. This technique of flattening a 2D data structure into 1D is applicable to our compute shader

solution later that will allow us to take full advantage of the performance of a simpler data structure. The *GetNeighbours* function serves to retrieve the particle IDs for a given position, by rounding said position to the nearest cell and calling *GetCell* to query the surrounding eight cells for their contents, which are appended to a list and then returned from the function.

After each step, we clear the grid and reinsert the particles with their new positions.

The other essential component before the actual simulation is the smoothing kernel logic, which we wrap in a class for ease of use; we define two values based on the particle smoothing radius, *POLY6* and *SPIKY*, which represent the constant values needed for applying the Poly6 kernel and the Spiky Gradient kernel, calculated as previously described:

```
POLY6 = (315.0f / (64.0f * Mathf.PI * Mathf.Pow(H, 9)));
SPIKY = (-45.0f / (Mathf.PI * Mathf.Pow(H, 6)));
```

The *Poly6Kernel* function applies the kernel based on two input vectors representing the positions of the two particles that are interacting, and returns a weight as a floating point:

```
public float Poly6Kernel(Vector2 pi, Vector2 pj) {
    Vector2 r = pi - pj;
    float len2 = r.sqrMagnitude;
    if (len2 > Hsqr || len2 <= 0) {
        return 0f;
    } else {
        return (POLY6 * Mathf.Pow(Hsqr - r.magnitude, 3f));
    }
}
```

The *SpikyGradient* function similarly applies the spiky kernel:

```
public Vector2 SpikyGradient(Vector2 pi, Vector2 pj) {
    Vector2 r = pi - pj;
    float len2 = r.sqrMagnitude;
    if (len2 > Hsqr || len2 <= 0) {
        return Vector2.zero;
    } else {
```

```
        float len = r.magnitude;
        r.Normalize();
        float term = (H - len) * (H - len) * SPIKY;
        return r * term;
    }
}
```

We then define the function that acts as our simulation step, performing the logic described here:

```
For each particle p:
    F_p = gravity
    v_p =  v_p + F_p * Δt
    r_p = r_p + v_p * Δt
    Enforce boundary condition

Generate neighbourhoods

For n solver iterations:
    For each particle p:
        ρ_p = 0
        For each neighbour n:
            ρ_p = ρ_p + mass * Poly6kernel(r_p, r_n)
        equationOfState = (ρ_p / restingDensity) - 1
        vectorSum = 0, magnitudeSum = 0
        For each neighbour n:
            gradient = spikyGradient(r_p, r_n)
            vectorSum = vectorSum + gradient
            magnitudeSum = magnitudeSum + ||gradient||^2
        magnitudeSum = magnitudeSum + ||vectorSum||^2
        λ_p = equationOfState / (magnitudeSum + ε) * -1
    For each particle p:
        Δp = 0
        For each neighbour n:
            s_correction = K * Poly6Kernel(r_p, r_n)^N
            Δp = Δp + SpikyGradient(r_p, r_n) * (λ_p + λ_n + s_correction)
        Δr_p = Δp / restingDensity
    For each particle p:
        r_p = r_p + Δr_p
        Enforce boundary condition
For each particle p:
    v_p = (r_p - x_p) / Δt
    Apply XSPH Viscosity & Vorticity Confinement
```

34	$x_p = r_p$

The positions *x* can then be translated to world-space coordinates and used to relocate the ghost objects that are then rendered as circular sprites.

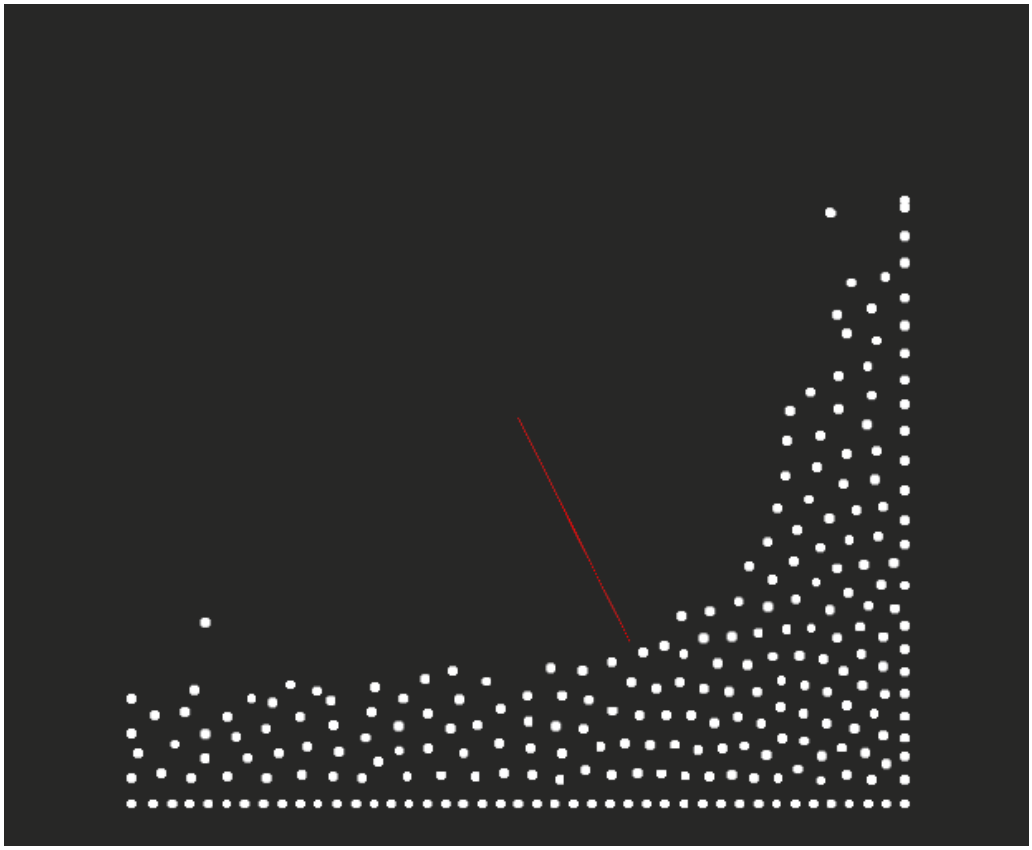

Figure 5: *The prototype 2D simulation, with white dots representing each particle and the red line showing the direction of gravity.*

With 256 particles and a gravity force that slowly rotates over time to create waves, the simulation operates at an average of around 11 FPS (~90 ms/frame). The result is a coherent body of fluid that exhibits surface tension, convection, and an even distribution of particles, except for near the borders where particles seem to form clusters possibly due to neighbourhood deficiencies when attempting to retain density.

3.2 Final 3D Implementation

With the concepts and governing formulae now proven in two dimensions, we moved to 3D and to a much more sophisticated form of neighbourhood generation. Before

generating the neighbourhoods, we optimise the neighbourhood grid for faster memory access by binning particles into cells and sorting the particles by their cell, and their position within each cell. This ensures that when neighbours are looked up from the particle properties array during the simulation logic, the memory accesses are contiguous rather than scattered. To do this on the GPU, we use compute shaders, which interface with *C# Unity* scripts to manage the execution of kernels in parallel. Each time a kernel is executed in parallel, it is given an ID that serves to identify the thread and, in our case, the particle on which we are operating. The kernels that optimise the grid are called in this order each update:

1	*BuildGridCS*	Reads the particle position from *ParticlesBufferRead* and calculates the 3D grid cell coordinate that this particle occupies and writes the coordinate-ID pair to *GridBufferWrite*.
2	*<Bitonic Sort>*	In a separate compute shader and script, the particles and their grid coordinates are sorted by grid cell, and written back into the buffer for the next kernel in the grid-optimiser compute shader.
3	*ClearGridIndicesCS*	Clears the elements of the *GridIndicesBufferWrite* buffer.
4	*BuildGridIndicesCS*	Establishes a range of particle IDs for each cell, encoding their start and end indices in *GridIndicesBufferWrite*
5	*RearrangeParticlesCS*	Sorts the particle array's IDs by copying the previous sort.
6	*CopyBuffer*	Applies the previous sort to the rest of the particle properties using the newly updated particle IDs.

With the particles allocated to cells and sorted, the main simulation shader's *GenerateNeighbours* function is called, which builds the neighbourhoods themselves into an array that allows subsequent steps to quickly get neighbouring particles' IDs to query their properties:

```
1   For a given particle p:
2       neighbourCount_p = 0
3       Determine the cell this particle occupies: gridXYZ = GridCalculateCell(r_p)
4       For each axis A of XYZ:
5            Determine the lower and upper bounds
6       For Z = lower Z bound to upper
7           For Y = lower Y bound to upper
8               For X = lower X bound to upper
9                   Calculate the cell ID at X Y Z -> G_cell
10                  Get start & end IDs for the cell: G_startend =
11  GridIndicesBufferRead[G_cell]
12                  For other = G_start to G_end
13                      covector = r_p - r_other
14                      if (||covector||^2 < Radius^2)
                            neighbours[neighbourCount_p++] = other
```

Where previous literature has explored the use of such sorts as the parallel radix sort and the counting sort, we introduce the bitonic sort [Batcher 68] [15]. The bitonic sorting algorithm is appealing to us in this instance because we wish to take full advantage of the GPU's parallel processing and this particular sort's sequence of comparisons is fixed and independent of the input data. The bitonic sort is of $O(\log^2(n))$ parallel time complexity (best-case, worst-case, and average), and thus scales well for tens of thousands of particles in a heavily parallelised sort.

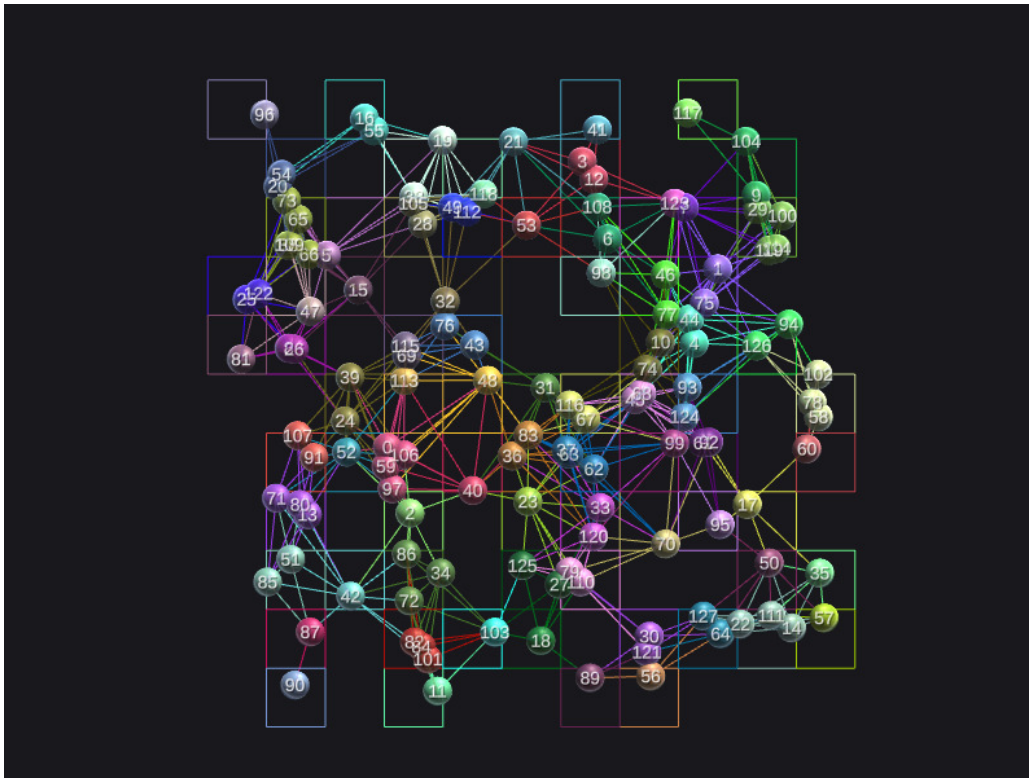

Figure 6: *A 2D demonstration of the neighbourhood generation, showing coloured cells, the particles they contain, and the relationship lines between particles determined to be within the same 3x3 neighbourhood.*

To prevent overflow, $neighbourCount_p$ contains the actual number of neighbours that a particle has, which allows us to use a fixed-size 2D array to store the neighbourhood information, leaving the unused elements in any neighbourhood as they were previously and simply stopping any loop over neighbours before reaching them, much like how if a particle ID is greater than the particle count it is ignored.
With the neighbourhoods constructed, the kernels that constitute our simulation can be called; essentially, each for-loop in the prototype's pseudo code is replaced with a kernel call to the main compute shader, before which we bind the relevant data. This binding of particle and neighbourhood data only involves a CPU-to-GPU transfer initially, as the full particle data is then kept on the GPU for the remainder of the simulation, including at the rendering stage where particle positions are passed to the shaders that draw them. Replacing the serial for-loop with parallel kernel calls offers one of the two largest performance boosts overall, the other being the absence of any data transfer between processors beyond the initialising frame.
We also take advantage of *D3D11*'s constant buffer feature, which allows the definition of semi-constant values to be done efficiently: uniform parameters for the simulation such as particle mass or gravity are passed using these *CBUFFER*s, which are

optimised for the occasional transfer of mostly-unchanging data, should we need to update any parameters, but not for the repeated sending of data to the GPU every frame.

3.3 Visualisation

To keep all relevant data on the GPU, we must be able to access the simulation data from the shaders and programs that govern the rendering of the fluid. Luckily, *Unity* provides us with a structured way to do this using the Scriptable Render Pipeline (SRP); it allows for structured use of explicit rendering commands, beyond what would be possible with a more traditional render pipeline, and allows us to implement our visualisation as a render feature, decentralised from any scene object and configurable on a global scale. The pipeline importantly lets us inject the drawing of the fluid at the correct stage during rendering, ensuring features like post processing and transparent objects interact with our fluid correctly.
To draw anything with multiple passes often requires the use of multiple render targets, which can be thought of as temporary textures or images that allow for the rendering of image data to buffers rather than directly to the screen. We take advantage of this to apply multiple stages of effects to the particle data to achieve a smoother, more realistic appearance. In the interest of performance, we use an RGBA render target with 8-bits per channel for all our colour render targets in this process, and use a packing technique to make use of three channels to encode depth in order to achieve higher precision.

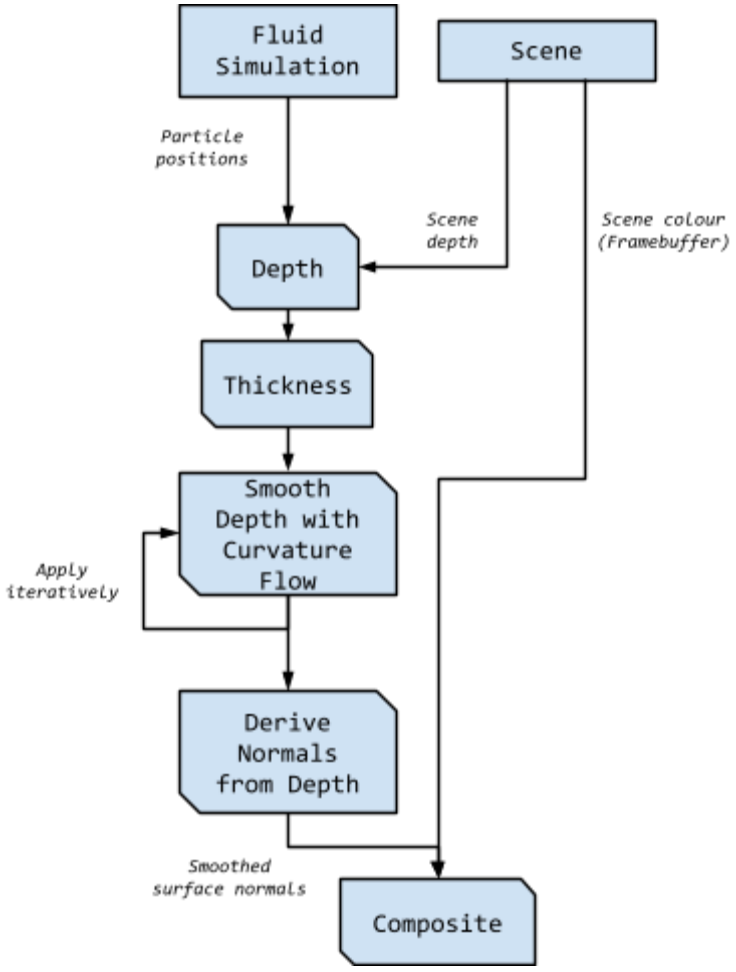

Figure 7: *A flow diagram of the rendering process, showing the shaders used and the data provided to them.*

We begin by taking the particle positions calculated by the simulation and passing them from a short vertex shader to a geometry shader, in which we construct simple quads pointed toward the camera. UV coordinates for the vertices are also generated, to enable us to generate the illusion of complex geometry in the next step.
In the fragment shader, we employ spherical impostors by constructing the image of a sphere using the UV coordinates generated previously to discard the pixel if it lies outside the radius of our circle, centred in the quad. Using the distance from the centre of the circle, we can generate the depth at each pixel as if it were the surface of a sphere, which is then added to the depth of the vertices. Two depths are required, one for the sorting of the particle quads and the other for the computation of normals later in the rendering process; these two depths are of different scales and are represented

with different encoding (logarithmic vs. linear), and while the former is output as a parameter from the fragment shader and managed by *Unity* automatically, we use the latter in the coming shaders to determine the normal direction at each point on each particle.
In order to achieve a high precision depth encoding, we devised a method of encoding a 24-bit depth value across three colour channels, achieved by simply splitting the less-significant 24-bit portion of a 32-bit floating point depth value into three 8-bit values of increasing significance and storing each in the R, G, and B channels of the render target. The result can later be decoded into a floating point value by using bit-level shifting.

We also construct the particle quads again in another, similar shader in order to estimate the “thickness” at each point — essentially, the amount of fluid between a pixel on the screen and the background. We store this in the alpha channel of the same buffer that the depth is written to, to save on the number of render targets we use overall. This later allows us to refract with variable strength (refractive index), as well as perform colour extinction accurately.
In an iterative process we then smooth the depth, generated prior, along a derivative of the change in depth across pixels according to curvature flow. This curvature flow process is parameterised and can be done in a variable number of iterations, depending on the balance between smoothness and performance desired. From this now-smoothed depth, we generate view-space surface normals from a weighted sum of the surrounding depth values at a given pixel, which we then use in conjunction with the existing camera framebuffer to layer the fluid on top of the scene, drawing the final surface lit using the lighting present in the scene. The fluid itself is parameterised such that the colour, refractive index, opacity, colour extinction, scatter colour, and scattering factor are able to be tuned to achieve the appearance desired.

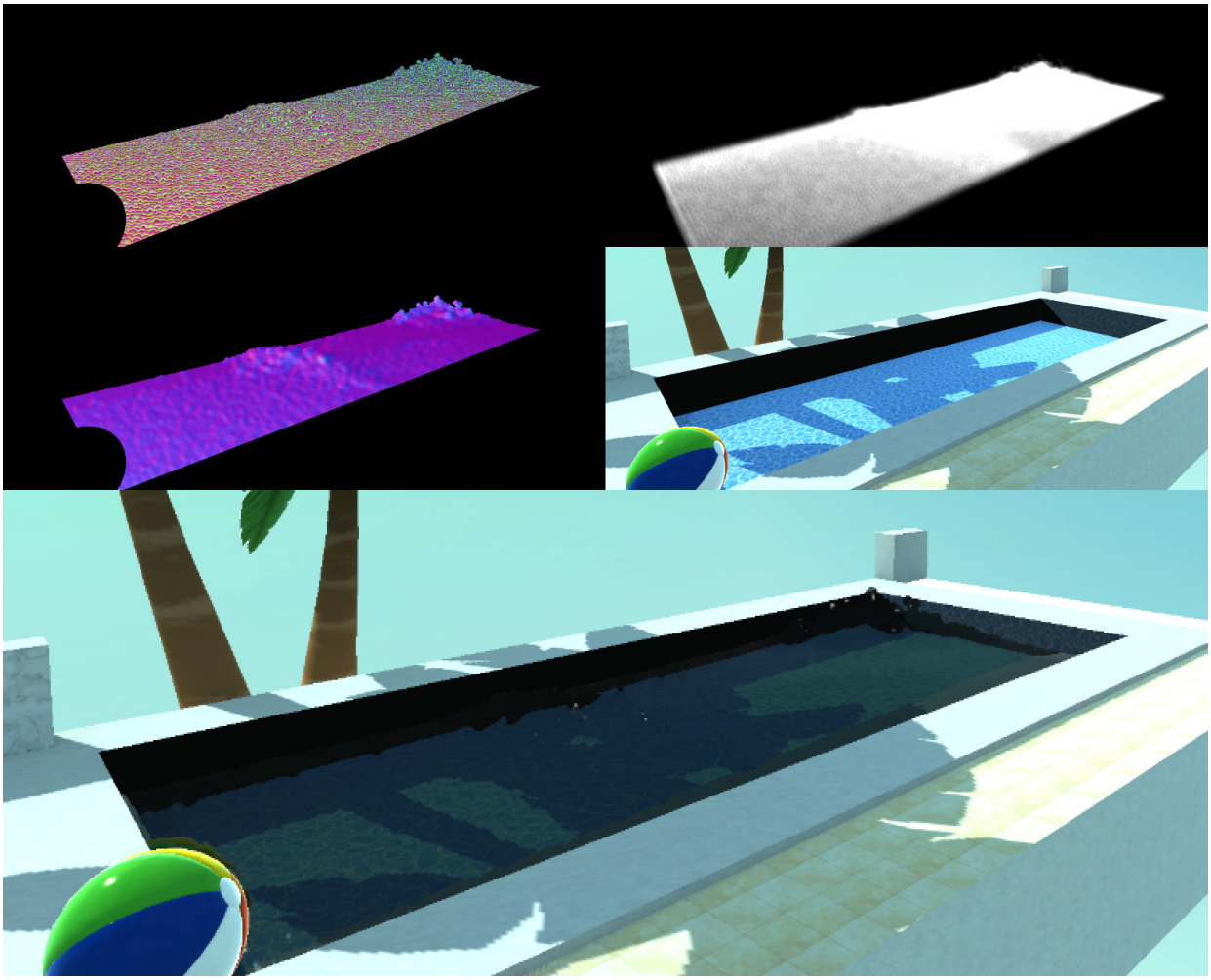

Figure 8: *From left-to-right, top-to-bottom: Fluid depth encoded into the RGB channels, thickness encoded into the alpha channel, surface normals estimated from smoothed depth, the framebuffer, and the final composite.*

The quality of the fluid surface depends on several factors independent of the resolution of the simulation, the chief ingredient being the smoothness and contiguity of the surfaces that should be created between particles or in other words the flatness of a fluid body at rest. To achieve this smoothness and create the illusion of a realistic fluid volume, most methods attempt to smooth the depth written to the framebuffer in some way or another. The original method we tried involved a naïve Gaussian blur of the depth data which, while fast, resulted in the blending of foreground and background objects, which looked completely unrealistic. The next attempt involved weighting the Gaussian blur's samples based on the difference in depth, so as to weaken blurring across borders. This also produced artefacts, as pixels toward the edges of particles would reject depth samples from outside the particle's silhouette while always accepting samples belonging to the same particle, meaning these

edgemost pixels would have a tendency to become unusually white in the smoothed depth, resulting in inverted puck shapes with a bevelling effect.

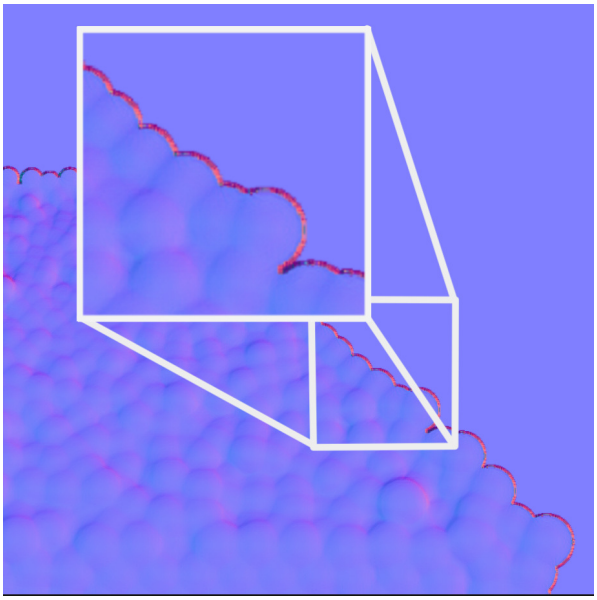

Figure 9: *Bevel artefacting on the edges of particle silhouettes caused by sample weights biased toward pixels closer to the centre of the particle.*

Additionally, even if this weighting issue were to be solved, the Gaussian blur technique would still result in the bubbly, gelatinous appearance seen in figure 7. An alternative would be to use a bilateral Gaussian blur, which preserves edges while blending low-frequency details; such filters are, however, inseparable, leading to poorer performance:

Image $(M \times N)$, filter kernel $(k \times k)$:
Unseparated convolution: $\mathbf{O(MNkk)}$
Separated convolution: $\mathbf{O(2 * MNk)}$

Figure 10: *Left: Lena, Right: Lena with an example bilateral filter applied, showing reduced high frequency detail but retention of edges.*

We could also employ a narrow-range filter as described by Truong and Yuksel [2018][11] which samples only those values within a given depth range, while treating depth values outside the range in a different manner to achieve the ideal flattening effect on colinear groups of particles while also avoiding the artefacts that surround discontinuities in depth. Whilst their solution corrects the issues found with naïve blurring methods, it introduces unwanted smoothness in the topographically more complex areas of the fluid body such as near splashes and fine mists of small clumps of particles, wherein particles that should appear unchanged by the blurring, as spheres, instead become flattened into circular discs facing the viewer.

An arguably more sophisticated method proposed by van der Laan [et al. 2009] [10], curvature flow, involves iterative correction of derivatives of differences in depth to minimise curvature across the depth map.While this method is notably more complex and computationally intensive, it produces results that are not only smooth and without the artefacting present in the alternative methods, but that are configurable to achieve a specific level of smoothness and performance. In this instance, we use a combination of this curvature flow method for depth smoothing with a separate weighted-sum method for constructing the normal vectors; we employ a method similar to the first steps of the narrow-range filter to discriminate depth samples when reconstructing normals from derivatives across the depth map, which helps to effect a gentle blurring on the normals to prevent avoid edges and high frequency details from causing jarring flickers in the end result.

4 Results & Evaluation

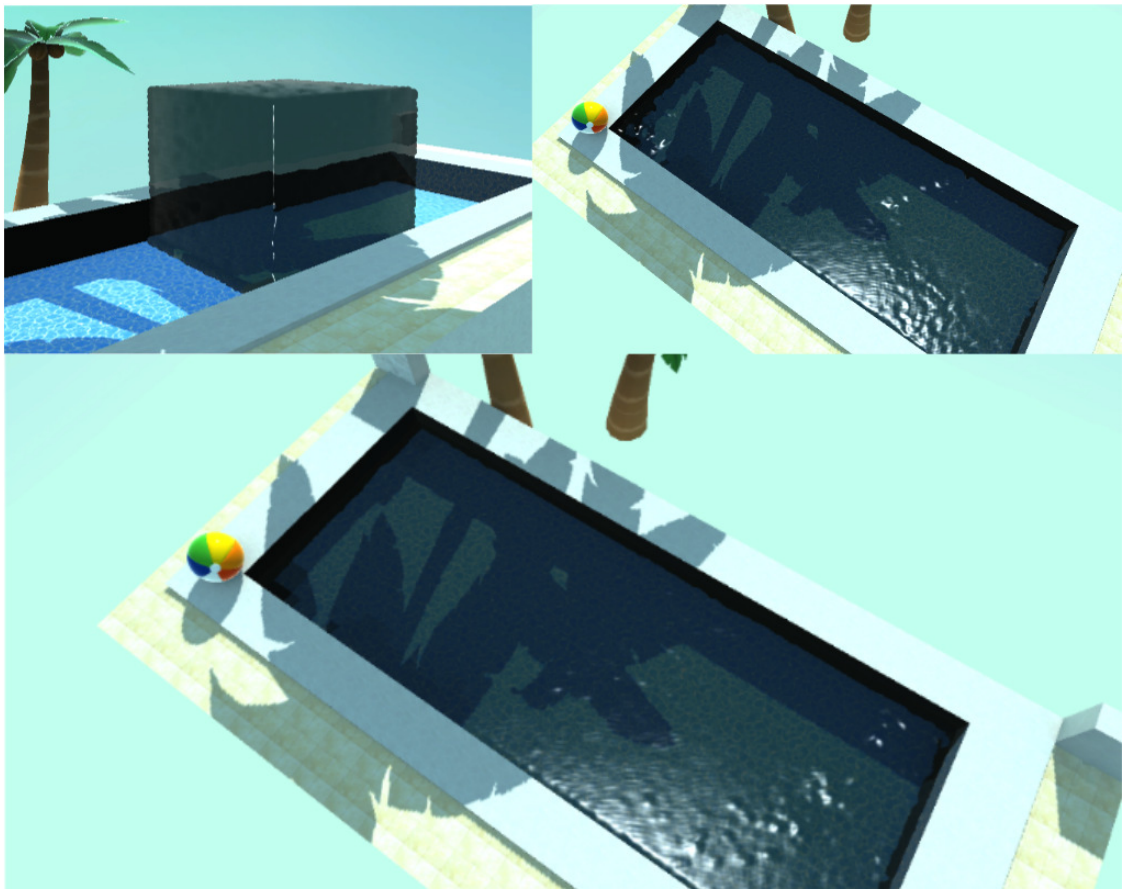

Figure 11: *Clockwise: The first few frames into a dam break scenario; small waves propagating through the now at-rest pool of water; another view of the pool, with even less motion than previously showing the smoothness of the fluid surface.*

Our method produces a smooth and visually stable end result, with a visibly smooth fluid surface that exhibits no flickering or obvious artefacting. The individual particles and their distribution is somewhat visible in the appearance of the fluid, although this looks relatively natural and believable compared to the raw, unsmoothed appearance.

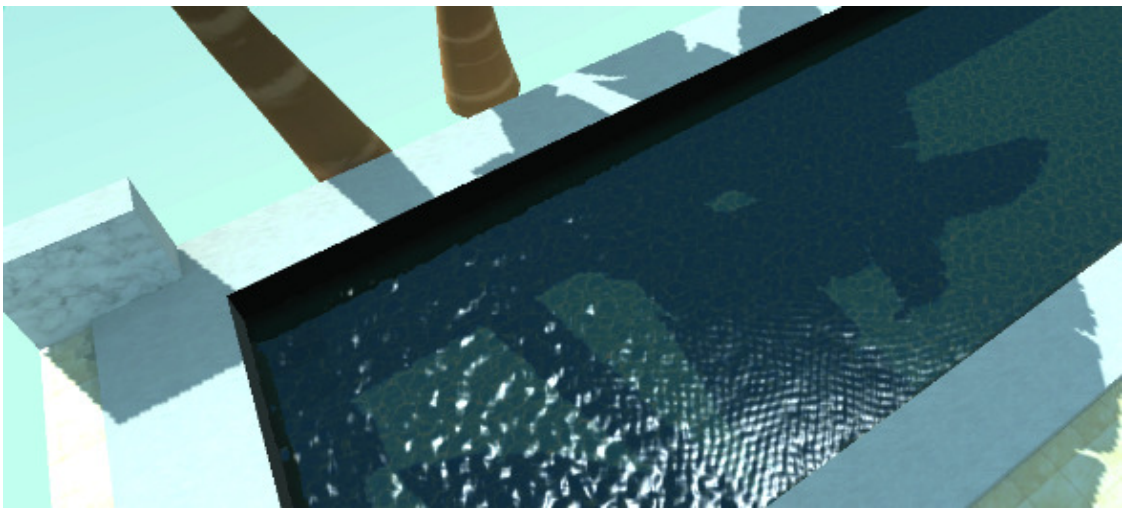

Figure 12: *The unsmoothed version of the pool scenario. Note the obvious silhouettes of each particle in the highlights on the surface.*

The fluid colour can be chosen as well as the opacity, which in turn affects the refractive index of the fluid, resulting in more distorted appearances of objects behind the fluid volume.

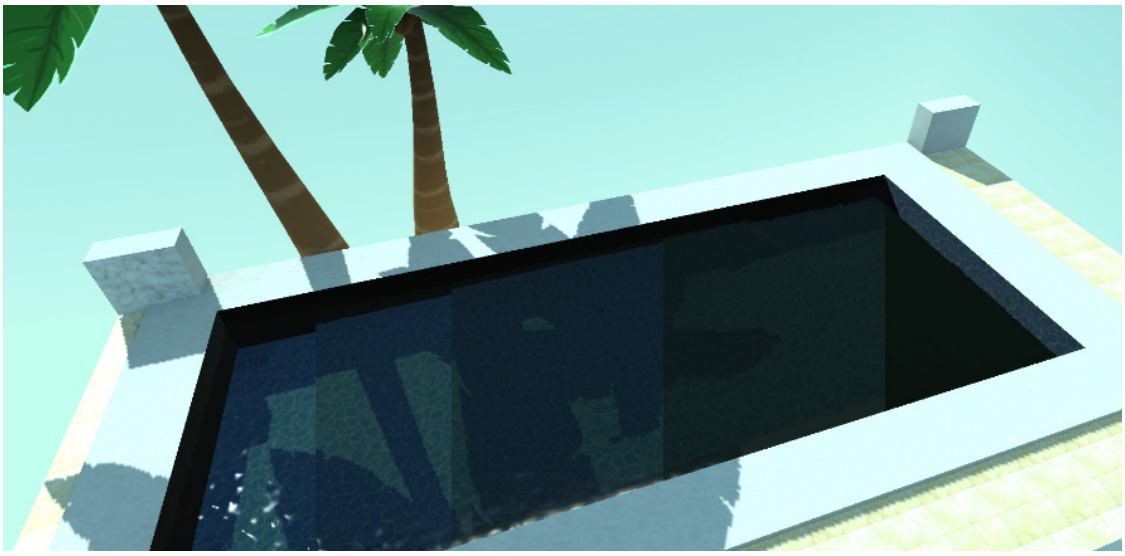

Figure 13: *From left to right, 0% fluid opacity to 100%, showing the effect on refraction and transparency.*

Additionally, both colour extinction levels and the scattering coefficient can be adjusted to achieve the desired colouration throughout the fluid, with the former controlling the rate at which certain wavelengths of colour are dissipated by increasing depth - a phenomenon commonly noted for giving oceans their blue-green appearance.

The latter coefficient controls the degree of scattering per unit of depth, shown in figure 11, that serves to mimic the absorption and scattering of incoming light.

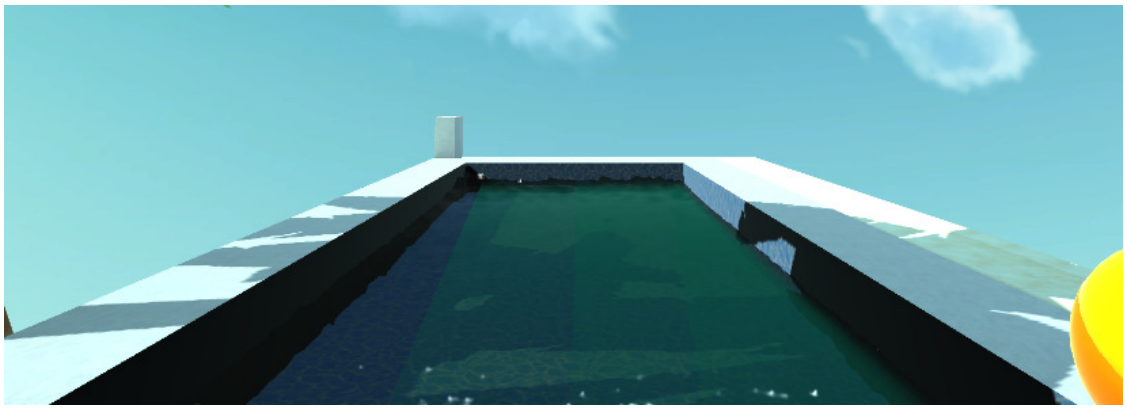

Figure 14: *From left to right, the effects of scattering coefficients: 0.0, 1.0, 2.0, and 3.0*

The parameters of the simulation itself can also be configured to achieve a variety of results; viscosity can be tuned to give greater or lesser cohesion between particles, the relaxation parameters can be set to achieve the desired balance of stability and realisticity, and naturally particle counts, radii, and mass are all completely parameterised. Particles can be created and destroyed or, more efficiently, pooled to create the illusion of creation and destruction; for example, a scene with a drain and a particle-emitting pipe could be created such that particles seem to constantly enter and leave the system.

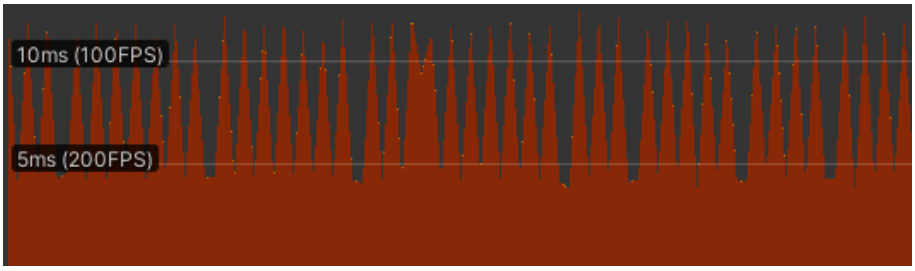

Figure 15: *A graph of the frame times in milliseconds*

Performance-wise, a fluid consisting of 32,768 particles in a lightly decorated scene with high quality shadows, the demo scenario runs at a consistent average of 150 FPS or a 7.5 millisecond frame time, on an *NVIDIA GeForce RTX 2080* GPU and an *AMD Ryzen 5 1600 six-core* CPU, with 16GB of RAM. This performance is steady across time and experiences no dips or spikes at any particular point in the simulation or camera angle.

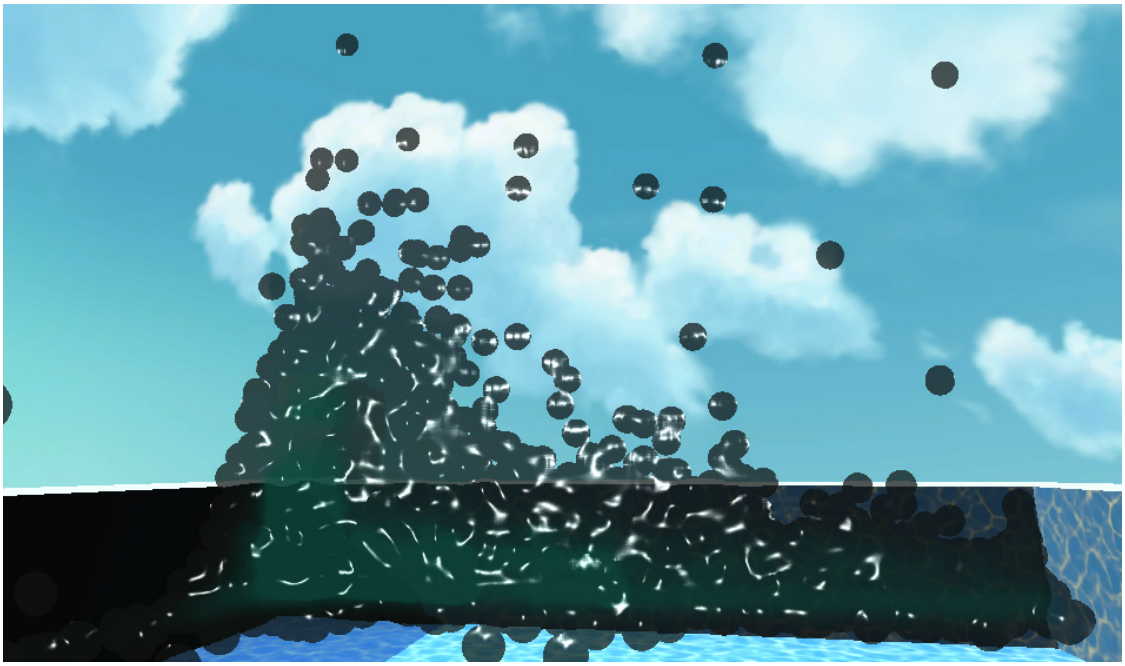

Figure 16: *The fluid body demonstrating cohesion and viscosity as it impacts the side of the pool.*

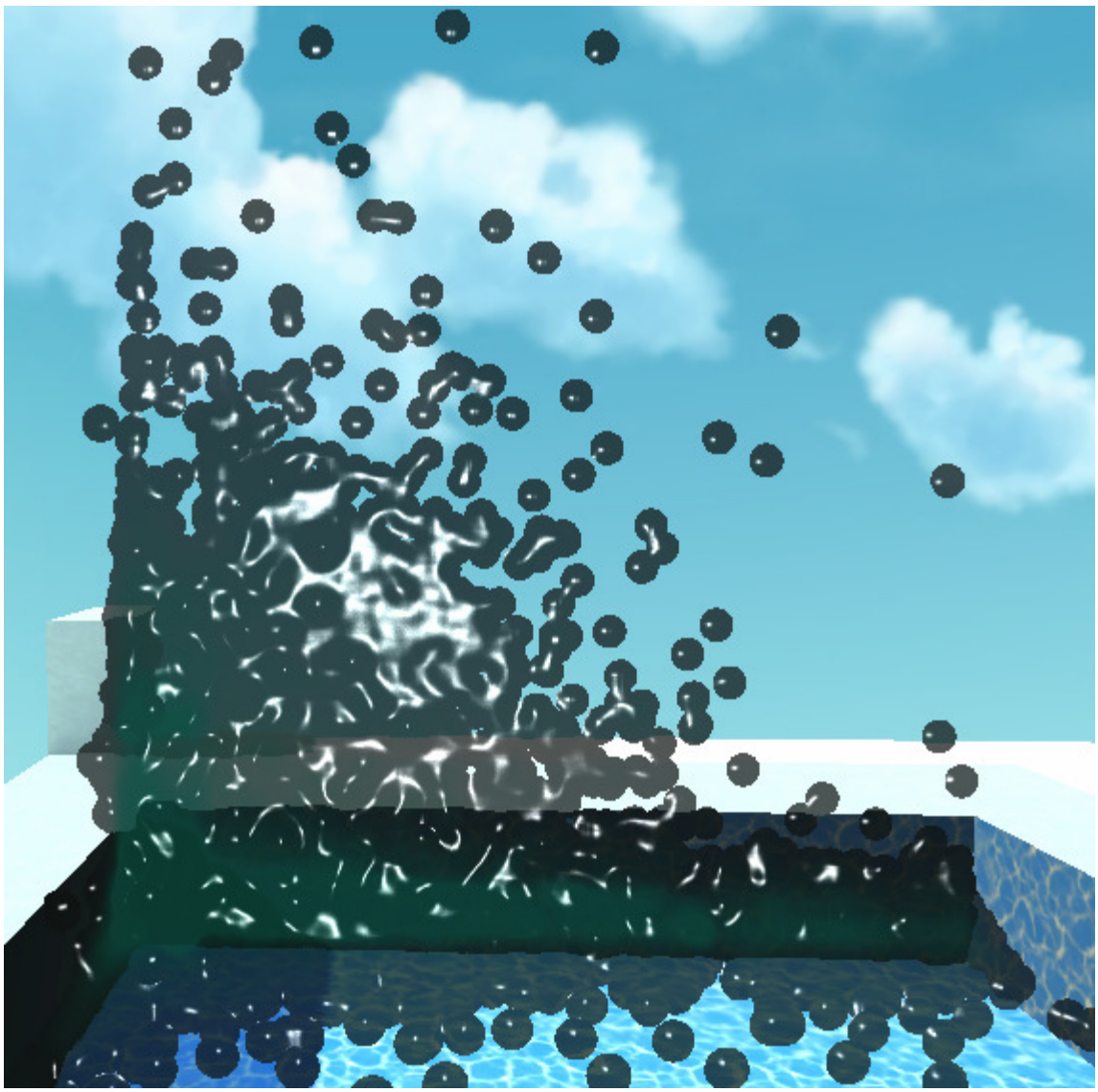

Figure 17: *Ditto.*

5 Future work

While a smooth appearance to the fluid-air boundary is desirable, many instances of real world fluids that may be replicated using fluid simulation have minute details that are not present in our method; this could be resolved by increasing the particle count and reducing the radius for a more resolute simulation, but this is impractically expensive. Instead, one could add a projected repeating noise texture, or even 3D Perlin noise-based textures as an additive normal map, to add faux detail to the surface of the fluid at minimal cost. Small clusters of particles could also be rendered as spray, perhaps by changing their geometry to be more elongated, and even substituting high-velocity particles' sphere impostors for animated splash sprites.

Our method does not support the addition of arbitrary colliding objects, nor the buoyancy involved in such an interaction. A number of ways exist to implement this, ranging from treating objects as fixed collections of less-dense particles and using the simulation results to inform the objects transform, to computing intersections between particles and a collision hull to generate buoyancy and other interactive forces on the objects and the fluid.

Given the implementation's reliance on a uniform grid for neighbourhood generation, increasing the bounds of a simulation hugely relative to the particle radius would be unreasonably expensive; instead, a smart system of divided bodies of fluid that use separate grids for representing collections of particles that are distant from one another could optimise the expansion of the simulation bounds by dividing up the total bounding area of the simulation into sub-grids that are independently simulated, dividing and unifying these sub-grids as necessary to maintain the impression of a single fluid.

While our simulation treats the entire fluid as a single material, the introduction of material properties on a per-particle level would allow for multiple phases of fluids, including the simulation of bodies of air that would surround a liquid, applying drag-related motion and convection of the fluid to add additional degrees of realism to the result. Such mixed-media fluid simulation is a field on the cutting edge of computer graphics research, including those simulation methods that aim to simulate solid debris and sediment with correct two-way coupling [Gao et al. 2018] [16], as well as integration of coarse, soluble material and soft-body objects [Yan et al. 2016] [17]. These recent advancements have been made mostly in offline simulation and rendering, however, and thus are probably some time away from being developed for interactive applications.

5 Conclusions

We prescribed previously that the result of our method be fast enough for interactive media and realistic in appearance, and to that end we have succeeded by devising a solution for the *Unity3D* engine that is capable of running tens of thousands of particles at a more than acceptable frame rate that has an appearance not too far from the æsthetics of published games with realistic art styles of the past decade or so, complete with image-based lighting and industry-standard shading techniques. There is, however, a great deal of room for immediate improvement, both in terms of performance and rendering; established techniques exist for optimising SPH through the use of such methods as adaptable particle radii that increase or decrease resolution depending on vorticity and velocity, as well as recent hybrid-grid-particle techniques able to simulate ten times the number of particles shown in this paper at similar frame times. A great deal of performance could also be saved by switching from the expensive curvature flow method to the less convoluted narrow-range filter Gaussian based techniques in conjunction with additional faux surface detail via Perlin noise and

the addition of cheap splash effect sprites to cover the spherical nature of particles involved in spray. To implement a means of mixing various fluid materials and phases in a real-time simulation is realistically beyond the scope of a paper such as this, however, and we will not consider its absence a failure in this instance, although it would have been possible to involve solid, buoyant objects using existing, thoroughly documented knowledge.
We show a reasonable level of artistic control for the final appearance of the fluid, allowing for it to be applied successfully to the typical needs of end-user projects, such as vast, oceanic bodies of aquamarine saltwater, foliage-dense brown silt-laden rivers, crystal-clear shallows, and even unusually large pools of blood, should the need arise.

6 Project Reflections

In my initial plan I had set a scope certainly too wide for both the time frame and my ability; I set out to create something lightweight with little impact on performance, such that it could be dropped into virtually any game or interactive project with little impact on the frame-time budget - a goal that failed to consider the resolution necessary to produce æsthetically pleasing, high quality results. While the end result does look as I had hoped and expected, the performance is still far from what I had envisioned; a standalone demo running at only 150 frames per second on a high-end GPU is not indicative of a solution that can be universally applied to any project.
As per Hofstadter's law, achieving a convincing and efficient fluid simulation alone took much longer than I had planned, even taking into account Hofstadter's law; this meant that I was unable to allocate time for the features I thought basic, such as interaction with arbitrary rigidbodies or a dynamically bounded simulation.

Despite not achieving all the results I had hoped, I am still cautiously impressed with this project at its climax; while it lacks a lot of planned features, those that I did implement were surprisingly complex and thus I find myself fulfilled with the unexpected challenges I did face. Though I have previously read scientific papers on computer graphics research, I have never before implemented anything from them, let alone an aggregate of multiple papers covering multiple approaches to the same problem, so it is safe to say that my ability to visualise such daunting mathematics and pseudocode and build a mental map of their operation has been fortified. In addition to now being confident in the construction and application of compute shaders, I feel much more equipped to tackle problems concerning parallel computation that I had previously had the opportunity to solve.

The experience of this project has certainly been of great value, and has provided me the opportunity to push myself in ways I hitherto knew not how.

7 Bibliography

[1] Monaghan, J. J. 1992. “Smoothed Particle Hydrodynamics.” *Annual Review of Astronomy and Astrophysics*. https://www.annualreviews.org/doi/pdf/10.1146/annurev.aa.30.090192.002551.

[2] Macklin, Miles, and Matthias Müller. 2013. “Position Based Fluids.” *ACM Transactions on Graphics*, 104, 32 (4): 1–12.

[3] Courant, R., K. Friedrichs, and H. Lewy. 1967. “On the Partial Difference Equations of Mathematical Physics.” *IBM Journal of Research and Development* 11 (2): 215–34.

[4] Solenthaler, B., and R. Pajarola. 2009. “Predictive-Corrective Incompressible SPH.” In *ACM SIGGRAPH 2009 Papers*, 1–6. SIGGRAPH ’09 40. New York, NY, USA: Association for Computing Machinery.

[5] Franklin, W. R. 2005. “Recent Developments in NEARPT3—Nearest Point Query in E3 with a Uniform Grid.” *15th Annual Fall Workshop on Computational*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2421&rep=rep1&type=pdf#page=59.

[6] Green, Simon. 2010. “Particle Simulation Using Cuda.” *NVIDIA Whitepaper* 6: 121–28.

[7] Hoetzlein, Rama C. 2014. “Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids.” *NVidia GPU Technology Conference.* https://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf

[8] Lorensen, William E., and Harvey E. Cline. 1987. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm.” *SIGGRAPH Comput. Graph.* 21 (4): 163–69.

[9] Müller, Matthias, Simon Schirm, and Stephan Duthaler. 2007. “Screen Space Meshes.” In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 9–15.

[10] Laan, Wladimir J. van der, Simon Green, and Miguel Sainz. 2009. “Screen Space Fluid Rendering with Curvature Flow.” In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, 91–98. I3D ’09. New York, NY, USA: Association for Computing Machinery.

[11] Truong, Nghia, and Cem Yuksel. 2018. “A Narrow-Range Filter for Screen-Space Fluid Rendering.” *Proc. ACM Comput. Graph. Interact. Tech.*, 17, 1 (1): 1–15.

[12] Müller, Matthias, David Charypar, and Markus H. Gross. 2003. “Particle-Based Fluid Simulation for Interactive Applications.” In *Symposium on Computer Animation*, 154–59.

[13] Desbrun, Mathieu, and Marie-Paule Gascuel. 1996. “Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies.” In *Computer Animation and Simulation ’96*, 61–76. Springer Vienna.

[14] Schechter, Hagit, and Robert Bridson. 2012. “Ghost SPH for Animating Water.” *ACM Transactions on Graphics*, 61, 31 (4): 1–8.

[15] Batcher, K. E. 1968. “Sorting Networks and Their Applications.” In *Proceedings of the April 30--May 2, 1968, Spring Joint Computer Conference*, 307–14. AFIPS ’68 (Spring). New York, NY, USA: Association for Computing Machinery.

[16] Gao, Ming, Andre Pradhana, Xuchen Han, Qi Guo, Grant Kot, Eftychios Sifakis, and Chenfanfu Jiang. 2018. “Animating Fluid Sediment Mixture in Particle-Laden Flows.” *ACM Transactions on Graphics*, 149, 37 (4): 1–11.

[17] Yan, Xiao, Yun-Tao Jiang, Chen-Feng Li, Ralph R. Martin, and Shi-Min Hu. 2016. “Multiphase SPH Simulation for Interactive Fluids and Solids.” *ACM Transactions on Graphics*, 79, 35 (4): 1–11.