An Optimised Pipeline for the Non-Polygonal Rendering of Particle-Based Fluid Simulations

George Fyles

BSc (Hons) Computer Games Technology, 2025

Faculty of Design, Informatics and Business
Department of Games Technology and Mathematics
Abertay University

Table of Contents

Table of Equations

Table of Figures

All figures without a reference were created by the author.

Table of Listings

Table of Tables

Acknowledgements

I would like to extend my deepest thanks to (soon-to-be Dr!) Erin Hughes, who has consistently pushed me to achieve my best work and whose level of expertise in and enthusiasm for graphics programming has served as a constant source of inspiration. It has been my honour to have her as my supervisor.

Thank you to Professor Ruth Falconer for filling in for Erin during her absence, and for her kind words of reassurance.

Dr James Threlfall generously provided invaluable feedback on a draft of the literature review chapter of this work; for this and his warm words of support, he has my utmost gratitude.

It would be remiss of me not to give a special acknowledgement to Jamie Buttenshaw, whose previous work in the subject area inspired much of my own, and who gave several helpful tips and insights along with much encouragement and enthusiasm. Jamie was also kind enough to assist with testing by running a couple of tests on his work PC.

Finally, thank you to my parents, grandparents, friends and loved ones who have supported me throughout my time at university and during the undertaking of this project - especially those who endured much ranting about compute shaders and besmirching of Microsoft and NVIDIA.

Abstract

Particle-based fluid simulation methods allow for unparalleled realism in the behaviours of fluids such as water in video games. They offer no inherent way of tracking surface geometry, however, thus various techniques for reconstructing and rendering the fluid surface have been investigated. One such method is to directly ray trace the isosurface extracted from a signed distance field (SDF) representing the fluid body without the use of an intermediary mesh, and various techniques have been proposed to reduce the computational costs involved.

This study aims to design and implement a pipeline for the non-polygonal rendering of particle-based fluid simulations, incorporating a few of the abovementioned cost-cutting techniques, and to evaluate its effectiveness at reducing rendering latency and memory consumption when compared to less advanced methods.

An application was developed in C++ with Direct3D 12 and DirectX Raytracing to investigate methods of optimising the direct ray tracing of SDF representations of particle-based fluid simulations. A novel narrow band technique, an optimised nearest neighbour search and the construction of a sparse brick set data structure were combined into a unified pipeline for achieving the abovementioned rendering technique. Two additional pipelines were implemented utilising less advanced and completely naïve methods, to compare the complex pipeline against. Unit testing was performed to gather performance and memory usage data, and visual fidelity was additionally investigated.

It was found that the complex pipeline offers an improvement over both the naïve and simple pipelines in most cases, with frame times and memory overhead scaling logarithmically as both the number of particles simulated and the resolution of the precomputed SDF increases. In the best case, the complex pipeline demonstrated a performance gain of nearly 6000%. Visual fidelity was found to be satisfactory.

This study concludes that the implemented pipeline is indeed a strong combination of techniques, effectively reducing rendering latency and memory consumption. The use of this pipeline could feasibly be applied to modern games; however, future research is needed to further optimise it and determine how it compares to polygonal rendering options.

Abbreviations, Symbols and Notation

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
AABB	Axis-Aligned Bounding Box
ALU	Arithmetic Logic Unit
BLAS	Bottom-Level Acceleration Structure
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CSV	Comma-Separated Values
DXR	DirectX Raytracing
GPU	Graphics Processing Unit
GSM	Groupshared Memory
GST	Grid Sphere Tracing
L2	Level 2
LSU	Load/Store Unit
MC	Marching Cubes
NNS	Nearest Neighbour Search
RT	Ray Tracing
SBS	Sparse Brick Set
SCF	Smoothed Colour Field
SDF	Signed Distance Field
SM	Streaming Multiprocessor
SPH	Smoothed-Particle Hydrodynamics
SVO	Sparse Voxel Octree
SVS	Sparse Voxel Set

Symbols and Notation

c_S	Smoothed colour field
∇c_S	Gradient of smoothed colour field
$\boldsymbol{x}_i$	Position of particle i
r_i	Radius of particle i
h	Radius of support/influence
$\phi(\boldsymbol{x})$	Signed distance to position $\boldsymbol{x}$

Chapter 1 Introduction

Impressive, visually appealing fluid has been a desirable feature in video games for years, especially with respect to realistic water; realistic visuals and environmental behaviour can increase a player's sense of immersion, thereby contributing to increased enjoyment and improved player retention.

Simulating and rendering true, interactive, large-scale 3D fluids in real-time is often considered too computationally expensive (Liu, *et al.*, 2023). Some modern games such as *Horizon Forbidden West* opt instead to extract pre-baked animation data from offline simulations to manipulate a plane formed of triangles (Malan, 2022). While producing visually impressive results, the player cannot interact with the fluid, potentially breaking immersion. Another widely used approach is to limit simulations to two dimensions, reducing computational complexity thus allowing for real-time reactivity. This approach is still limited in realism, however, due to a lack of realistic 3D details such as true-to-life splashes or submerged objects affecting turbulence (Kellomäki, 2012). Evidently, research into simulating and rendering true 3D fluids in real-time is worthwhile.

Smoothed Particle Hydrodynamics (SPH) is a popular and widely researched method of simulating life-like 3D fluids, with its use for simulating water in computer graphics introduced by Müller, Charypar and Gross (2003). SPH is a particle-based solver, whereby the fluid body is discretised into many individual particles, representing the fluid's spatial differential operators and spatial field quantities (Koschier, *et al.*, 2022). Navier-Stokes equations, which describe the motion of viscous fluids, are then calculated for each particle independently, and collectively the particles describe the behaviour of the fluid body. Figure 1 illustrates the concept of describing a fluid as a collection of discrete particles.

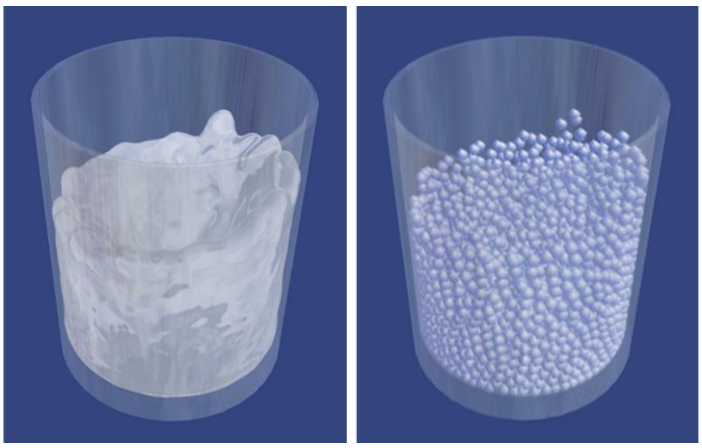

Figure 1 - A volume of fluid is rendered as a continuum (Left) and as discretised particles (Right) (Müller, Charypar and Gross 2003).

The process of simulating each particle independently is inherently parallelisable, which has allowed researchers to efficiently simulate millions of particles utilizing the massively parallel hardware capabilities of modern GPUs (Liu, et al., 2023). While promising, particle-based fluid simulations such as SPH do not come without drawbacks. SPH deals entirely with lists of arbitrary particle positions and other accompanying data, with no inherent process for tracking the geometry of the fluid's surface.

Various techniques for reconstructing and rendering the fluid surface have been investigated, but achieving this at real-time framerates for millions of particles remains an ongoing challenge. One means of rendering the surface while retaining small details in the geometry is to sphere trace (Hart, 1996) the signed distance field (SDF) representing the fluid body. The SDF is a function returning the signed distance to the surface from a given point in space and is computed by combining the signed distances to all nearby fluid particles. Evaluating the SDF is a computationally expensive process when many particles are involved, and sphere tracing requires it to be re-evaluated many times per ray, thus for many years this has not been considered feasible for video games.

Recent years have seen much research investigating the reduction in computational costs for these processes, including the use of a two-level uniform spatial grid to limit the particles considered during evaluation to those in a *narrow band* region around the surface (Nishidate and Fujishiro, 2024), and organising particle data in memory in a spatially coherent manner to allow for a fast neighbourhood search (Liu, *et al.*, 2023). The cost of SDF re-evaluation during sphere tracing can be mitigated by precomputing the SDF, storing discrete values in a 3D volume to be sampled. This has been employed in games such as *Dreams*

(Evans, 2015) and *Claybook* (Aaltonen, 2018), proving the feasibility of deformable SDF geometry in real-time. *Dreams* uses a sparse brick set (SBS) data structure to reduce the increased memory usage introduced by the storage of SDF values, a useful quality in memory-constrained scenarios such as the simulation of millions of fluid particles.

1.1 Aim, Research Question and Objectives

This study aims to consolidate the abovementioned techniques into a single pipeline for the non-polygonal rendering of particle-based fluid simulations, and to evaluate both its effectiveness at reducing the rendering latency and memory consumption when compared to less advanced methods and its feasibility for use in modern games. The study will attempt to answer the following research question:

- ❖ "How effective is the combination of a novel narrow band technique, an optimised nearest neighbour search and a sparse brick set at improving the performance and memory usage of the non-polygonal rendering of particle-based fluid simulations?"

The objectives of the study are:

- To review current techniques involved in reconstructing and rendering surfaces of SPH simulations.
- To develop a pipeline for non-polygonal rendering incorporating some chosen techniques, and to develop accompanying naïve and less advanced implementations.
- To gather performance and memory usage data to evaluate the effectiveness of the pipeline over the simpler methods. Visual fidelity will also be evaluated.

1.2 Overview

Chapter 2 will review literature surrounding the efficient reconstruction and rendering of implicit surfaces such as those of SPH simulations. Chapter 3 will cover the implementation of the artifact developed as part of this research, including the testing methodology. The test results are presented in Chapter 4, with a discussion and analysis provided in Chapter 5. Chapter 6 concludes the research with a summary and suggestions for future work.

Chapter 2 Literature Review

This chapter will review relevant techniques for rendering surfaces defined by collections of particles, with a focus on real-time use for fluid simulations.

2.1 Fluid Surface Reconstruction

Broadly speaking, techniques to reconstruct the surface of a fluid body fall into one of two categories: non-polygonal and polygonal methods. Most methods begin by using particle data to evaluate a scalar field, to find an isosurface (a level set of the continuous scalar field). Polygonal methods create an intermediary representation of the fluid body constructed of polygons conforming to the isosurface; a mesh to be rendered through conventional rasterization techniques. In contrast, non-polygonal methods render the isosurface directly.

2.1.1 Scalar Field Evaluation

In both polygonal and non-polygonal approaches to surface reconstruction, an isosurface is extracted from a scalar field. A scalar field is a "function associating a single number to each point in a region of space" (Wikipedia, 2024). Various scalar fields have been proposed.

Müller, Charypar and Gross (2003) suggest using a smoothed colour field (SCF), also known as a density field. A colour field is a binary representation of particle positions, returning values of 1 at particle positions and 0 everywhere else. The SCF is a smoothed version, shown in Figure 2, which sums contributions from surrounding particles, accounting for density and mass taken from the SPH simulation:

$$c_S(\boldsymbol{x}) = \sum_i m_i \frac{1}{\rho_i} W(\boldsymbol{x} - \boldsymbol{x_i}, h) \tag{1}$$

where $\boldsymbol{x}$ is the position in space for which the SCF is being evaluated, m_i is the mass of particle i, $\boldsymbol{x_i}$ its position and ρ_i the density. $W(\boldsymbol{x}, h)$ is a smoothing function with support radius h. Smoothing functions are discussed later in this section. This scalar field comes with the drawback that identifying the isosurface is not trivial and requires manual tweaking of a highly sensitive threshold value, discussed in Section 2.3.

Zhu and Bridson (2005) instead propose use of a signed distance field (SDF), which returns the signed distance to the surface from any point in space. In an SDF representation of a fluid body, values are positive outside the fluid body, 0 at the surface and negative inside. Therefore, in contrast to an SCF, identifying the isosurface is straightforward as it is simply the zero level set of the SDF. A comparison of the two types of scalar field can be seen in Figure 2.

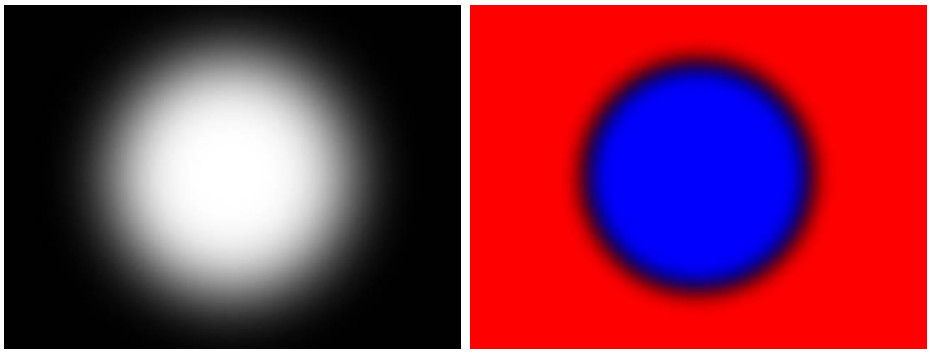

Figure 2 - SCF (left) and SDF (right). For the SCF, white denotes a field value of 1 and black denotes 0. For the SDF, red denotes a positive value, black 0 and blue a negative value.

The SDF is evaluated based on the principle that the signed distance ϕ from position in space $\boldsymbol{x}$ to a single particle i with radius r_i is given by:

$$\phi(\boldsymbol{x}) = |\boldsymbol{x} - \boldsymbol{x_i}| - r_i \qquad (2)$$

Similarly to SCF evaluation, contributions from surrounding particles are combined and smoothed by a smoothing function. In other words, each signed distance $\phi(\boldsymbol{x})$ is itself an SDF, and each of these are blended by the smoothing function to create a final SDF.

Due to the ease of locating the isosurface, SDFs lend themselves well to direct raytracing techniques such as sphere tracing, discussed in Section 2.2, and have been widely used for this purpose, including by *Dreams* (Evans, 2015), *Claybook* (Aaltonen, 2018), Söderlund, Evans and Akenine-Möller (2022) and Liu *et al*. (2023).

Scalar field evaluation makes use of a smoothing function, generally a *smooth minimum function* which smoothly converges towards the minimum function as a parameter approaches infinity, allowing the blending of SDFs of multiple objects. *Dreams* (Evans, 2015) uses the function, shown in Figure 3, which blends over a fixed radius of influence h. For two signed distance values a and b, the function SmoothMin is defined as:

$$\varepsilon = \max(h - |a - b|, 0) \tag{3}$$

$$\text{SmoothMin}(a, b) = \min(a, b) - \frac{0.25\varepsilon^2}{h} \tag{4}$$

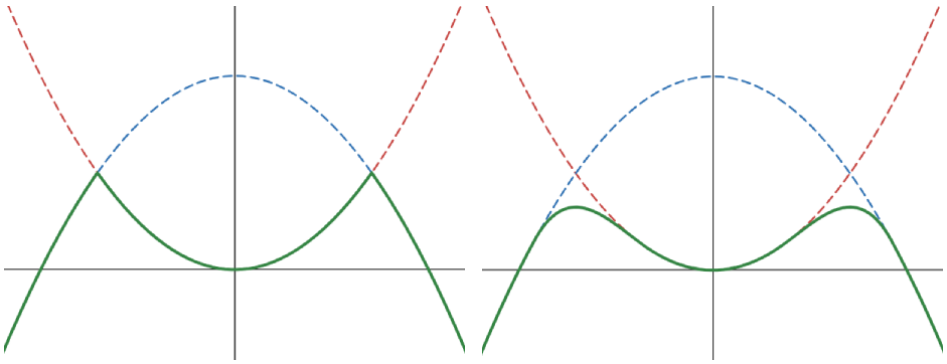

Figure 3 - Regular min function (Left) and smooth min used by Evans (2015) (Right).

Many similar functions have also been proposed; however, the impact of the chosen function is not a focus of this research, thus the above function will be selected for its ease of implementation.

2.1.2 Polygonal Methods

Polygonal methods, in which an intermediary mesh representation of the fluid body is constructed, are generally based on the Marching Cubes (MC) algorithm and its derivatives.

First presented by Lorensen and Cline (1987), the MC algorithm evaluates a scalar field at the vertices of *cubes* forming a uniform spatial grid. Each vertex is assigned a value of 1 where the field value equals or exceeds a given threshold value, and a value of 0 otherwise. With this information a surface topology for each cube is identified.

Given Boolean values for each vertex of a cube, there are $2^8 = 256$ possible topologies. By considering two symmetries of the cube, these cases can be reduced to 15 basic topologies, shown in Figure 4.

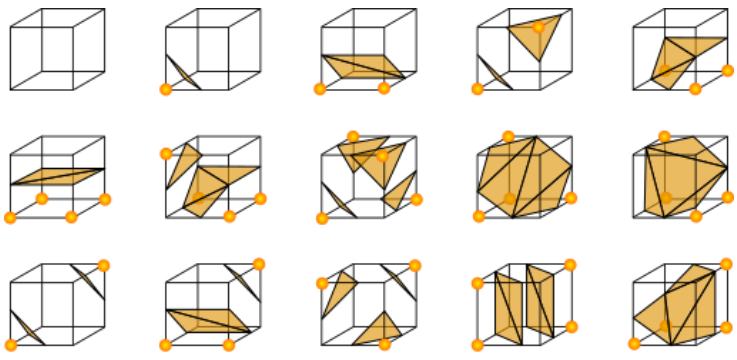

Figure 4 - Marching Cubes basic topologies (Fisher, 2014).

Transformations of these base cases produces 256 configurations, and a lookup table of surface-edge intersections is created, each entry containing the edges intersected for each case. The index for each case in the table consists of an 8-bit value, with the bits representing each vertex.

The algorithm *marches* from one cube to the next, indexing into the table to find which edges the surface intersects. For each intersected edge, a triangle vertex is calculated by linearly interpolating along the edge using the scalar values associated with that edge. The overall generated mesh is the union of all triangles generated by the algorithm. This may then be rendered through traditional rasterization techniques.

Improvements to the original algorithm have been proposed to ensure correct mesh topology, including extending the number of basic topologies to 33 and using trilinear interpolation (Zhang *et al*., 2023). Methods for parallelising the algorithm to run efficiently on GPUs have also been introduced, for example by Liu *et al*. (2023).

While MC-based methods are widely used for rendering particle-based fluid simulations, Söderlund, Evans and Akenine-Möller (2022) point out that limiting surface topologies to a set of base cases limits the accuracy of the surface obtained. Evans (2015) further explains that generated meshes are often undesirably dense and describes the edges produced as “mushy”.

2.1.3 Non-Polygonal Methods

Non-polygonal methods instead render the fluid surface directly, without an intermediary mesh representation.

Screen space solutions such as point splatting and its derivatives are one such class of non-polygonal method (Müller, Charypar and Gross, 2003), (Quispe and Paiva, 2022), (Liu, et al., 2023). These methods work by directly rasterizing the particles into a depth buffer as points, before blurring the depth buffer by applying smoothing convolution kernels. An example of point splatting can be seen in Figure 5; while overall creating the appearance of a smooth surface, some features

(as shown in the enhanced view of part of the surface) show unrealistic surface behaviour.

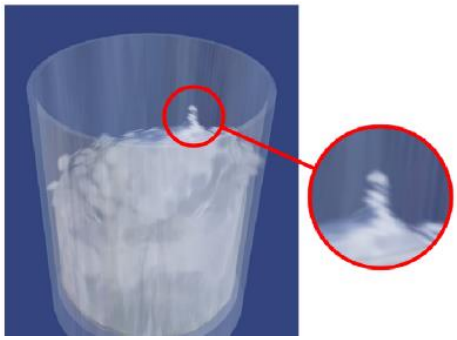

Figure 5 - Point Splatting (Müller, Charypar and Gross, 2003). Note the artifacts present.

A technique allowing for more visually pleasing results is to directly render an SDF through ray tracing. Recent examples of this approach include Söderlund, Evans and Akenine-Möller (2022) and Liu *et al*. (2023), and indeed was directly applied in several games, including *Dreams* (Evans, 2015) and *Claybook* (Aaltonen, 2018).

Perhaps the most widely known method of ray tracing an SDF is through sphere tracing (Hart, 1996). This algorithm is a form of ray marching which quickly converges to find the point of intersection between a ray and the implicit isosurface. Sphere tracing, shown in Figure 6, works by sampling the SDF at points along a ray, with the step size adaptively selected to equal the evaluated signed distance to the isosurface. This is based on the notion that the absolute distance can be considered the radius of an *unbounding sphere* – a volume guaranteed not to enclose any part of the isosurface. Once the sampled absolute distance falls below a chosen threshold value, the point of intersection is found. Pseudocode for the algorithm is shown in Listing 1.

Given SDF ϕ, ray R, max traversal distance D,

```
1. t = 0
2. While t < D:
    2.1.    distance = ϕ(R(t))
    2.2.    If distance < threshold value:
        2.2.1.  Return t        // intersection found
    2.3.    t += distance
3. Return null          // no intersection found
```

Listing 1 - Sphere tracing algorithm pseudocode.

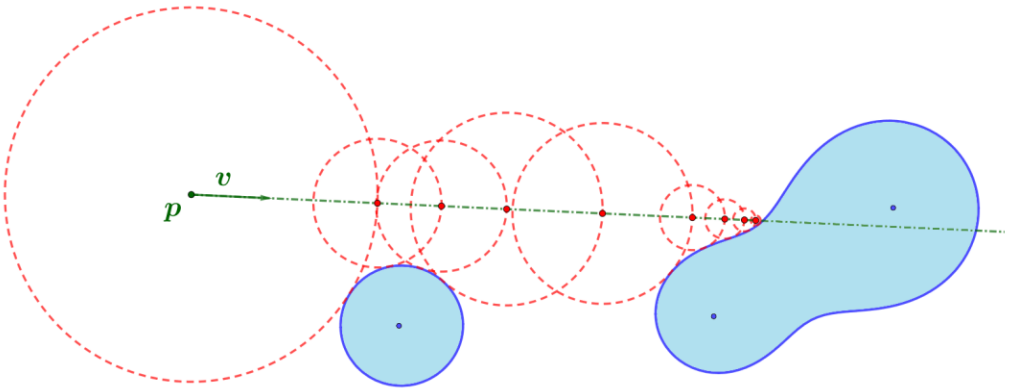

Figure 6 - Sphere tracing diagram (Bálint, Valasek and Gergó, 2021). Step sizes equal radii of unbounding volumes, shown in red.

Alternative methods of finding the intersection between a ray and the isosurface have also been developed. Söderlund, Evans and Akenine-Möller (2022) propose an analytic method where the trilinear interpolation of eight signed distances arranged in a cube is used to form a cubic polynomial, which after substituting in the ray components can be solved analytically or numerically to find the ray-isosurface intersection. This proposed method was found to be significantly faster (~41% decrease in rendering latency in some cases) than sphere tracing in all scenes, however results show sphere tracing to still be sufficient for achieving interactive frame rates in simple, open scenes such as a fluid simulation.

2.2 Discretisation and Data Structures

When ray tracing an SDF, a naïve implementation would evaluate the function at every step along every ray. A more effective solution is to precompute the SDF, discretising it into a grid of voxels, and store the result as a 3D texture to be sampled in the ray tracing step. There are several options to store and traverse such data.

The simplest solution is to generate and store a dense 3D texture of a fixed size representing the entire scene, storing signed distance values in all voxels, even those spatially far away from the surface. An enhanced version of this method is employed by the game *Claybook* (Aaltonen, 2018), which uses additional mip levels (lower resolution versions of the same 3D texture) encoding greater

distances per voxel to accelerate sphere tracing by adaptively selecting the sampled mip level.

Sparse voxel octrees (SVOs) are a more advanced method for storing and traversing voxel data. Crassin *et al.* (2009) propose ray tracing against such a structure, with nodes either pointing to 16^3 or 32^3 *bricks* of voxels or storing a constant value for homogenous regions of the scalar field. Bricks are stored separately in a 3D texture known as the *brick pool*. Child nodes are efficiently accessed through the pointer to the first child by storing sibling nodes contiguously in memory. All nodes may point to a brick representing the area of the volume corresponding to the node, meaning non-leaf nodes encode coarser resolutions, allowing for mip-mapping to produce innately anti- images aliased (meaning edges appear smooth) when rendering the volume. Furthermore, with static geometry in mind, the paper employs a method for streaming bricks in and out of memory in a view-dependent way, where the GPU informs the CPU which missing bricks are required to be loaded for the next frame.

Laine and Karras (2010) similarly present an SVO, with each node representing a single voxel. The status and properties of child nodes are inferred from bitmasks stored by parent nodes, meaning leaf nodes need not be stored explicitly. Similarly to Crassin *et al.*, the authors stream voxels in and out of memory by splitting the tree into sub-trees called *blocks*, which are loaded and unloaded as necessary. SVOs are shown to be efficient at storing and rendering voxel-based static geometry, however the updating of tree structures is not easily parallelisable and thus cannot take full advantage of GPU hardware. For this reason, SVOs are less well suited to dynamic geometry such as fluid simulations, where structure updates are required every frame.

The game *Dreams* (2015) employs a slightly different strategy, called a sparse brick set (SBS). This scheme is similar to that of Crassin *et al*. (2009), in that it stores bricks of SDF values in a brick pool. However, instead of using an octree, axis-aligned bounding boxes (AABBs) are created for each brick intersecting the surface, from which a bounding volume hierarchy (BVH) is created to accelerate ray tracing. When an AABB is intersected, the corresponding brick is accessed for rendering through a lookup table.

Söderlund, Evans and Akenine-Möller (2022) carry out a comparison of ray traversal times for all the abovementioned methods. It was found that in all scenarios, the fastest method (~34-48% decrease in rendering latency over the next-best procedure, depending on the scene) was a version of SBS, called sparse voxel set (SVS), where each AABB corresponds to a single voxel. This is because using one voxel per AABB has the effect that less work is required in the intersection shader.

For simpler scenes, the dense 3D texture method, which the authors call grid sphere tracing (GST), performed next fastest, likely due to most rays accessing similar data. However, SBS and SVO both followed closely.

In more complex scenes, GST was instead the slowest, with SBS taking second place after SVS. SBS and SVS perform the best as they take advantage of DirectX Raytracing (DXR) hardware for BVH traversal.

In terms of memory usage, SVS has a massive overhead as additional data is stored for every voxel. In contrast, SBS and SVO both use the least memory as they store less data per voxel and better exploit sparsity. GST memory usage is constant due to the fixed size of the 3D texture.

Fluid simulations are highly dynamic, thus the complexity of a scene at any given time is not guaranteed to be simple. Additionally, simulations involving many particles and calculations are memory intensive, therefore memory usage should be reduced wherever possible. For this reason, SBS seems a good compromise between GST and SVS. Furthermore, DXR hardware allows for fast BVH construction and traversal, used by the SBS, compared to the slower, hard-to-parallelise construction of octrees.

Quispe and Paiva (2022) present a suggestion for the enhancement of discretised scalar fields. In their research, the number of contained particles in each cell of a spatial grid is compared to the average count to approximate a colour field similar to that used by Müller, Charypar and Gross (2003) stored in a 3D texture. They then smooth the 3D texture using 3D box and Gaussian filters to reduce surface discontinuities and blockiness. This is an interesting technique and demonstrates how smoothed values could be found without relying on hardware interpolation.

2.3 Narrow Band Methods

As discussed in Section 2.1, most surface reconstruction methods require scalar field evaluation at points in space distributed regularly upon a uniform spatial grid. This stage is computationally expensive and is often the most time-consuming step in the surface reconstruction process due to the large number of evaluations necessary when using a large grid, especially when evaluating across the entire grid (Nishidate and Fujishiro, 2024). To reduce the computational cost, the evaluation can be limited only to those points in the spatial grid which lie within a *narrow band* region conforming to the fluid surface, an example of which is shown in Figure 7. Several techniques to identify and make use of this narrow band region have been explored to reach real-time framerates.

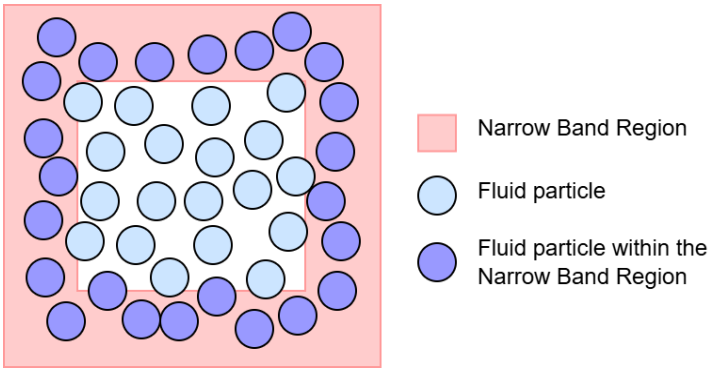

Figure 7 - Example of a narrow band region around the fluid surface.

Müller, Charypar and Gross (2003) identify surface particles using the smoothed colour field (SCF) c_S defined in Equation 1. A particle is a surface particle if the magnitude of the vector extracted from the gradient field ∇c_S at the particle's position is greater than a chosen threshold value. The expression for particle i is as follows:

$$|\nabla c_S(\boldsymbol{x}_i)| > l \tag{5}$$

This is where $\boldsymbol{x}_i$ is the position of the particle and l is the threshold value. ∇c_S returns surface normal vectors, oriented inwards towards the centre of the fluid body, decreasing in length as $\boldsymbol{x}_i$ approaches the centre. Therefore, the positions of particles at the surface return the longest vectors.

In a uniform spatial grid, cells containing the identified surface particles are then chosen to begin traversing the grid using the Marching Cubes (MC) method. This method comes with serious drawbacks, however. The threshold is highly sensitive – if set too low, some non-surface particles may be misclassified

as surface particles, and likewise a threshold that is too high may fail to capture all surface particles. Detecting isolated elements of the fluid body, such as splashes, also presents challenges requiring extra processing to address (Yang and Gao, 2020). Furthermore, this method generally only identifies the outermost layer of particles. Several inner layers of particles often lie within the radii of influence of the outermost particles and should be taken into account during scalar field evaluation (Akinci *et al.*, 2012).

Akinci *et al.* (2012) worked to parallelise the process to run efficiently on GPUs. Surface particles are determined in a pre-processing step as above, with the additional criterion that particles with less than 25 neighbours (a fixed value chosen by the authors) are also considered surface particles in order to detect remote regions such as splashes.

Axis-aligned bounding boxes (AABBs) of side length $2r_i$ are then constructed for all surface particles, and all grid vertices within these AABBs are determined to be surface vertices. This stage is performed sequentially to avoid race conditions.

To account for lack of inner particles within the radius of influence, a third stage determines particles contributing to the scalar value at each surface grid vertex. This is achieved by constructing more AABBs, this time for each surface vertex, of side length matching the radius of influence. The scalar field can then be evaluated at each surface vertex using the particles within these AABBs, to be used in the MC algorithm.

While this process improves upon earlier methods, it requires multiple stages of AABB construction and is not fully parallelised, due to the limitations of the SCF method described. A fully parallel method which can easily and accurately determine a narrow band of all particles contributing to the isosurface would be preferable.

Yang and Gao (2020) presented a novel technique for more accurately detecting surface particles which is straightforward and fully parallel. All particles are mapped into the cells of a uniform spatial grid, with each cell tracking the number of particles it contains. With this, a cell is simply identified as a surface cell if it is

not empty and at least one of the neighbouring 26 cells is empty. The process is shown in Figure 8.

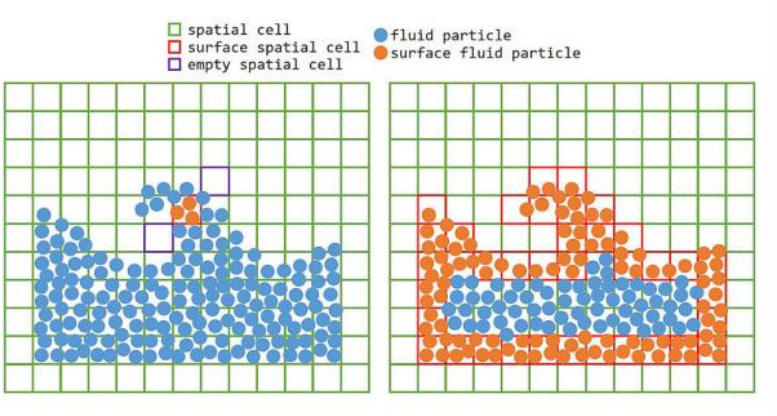

Figure 8 - Identification of a surface cell (Left) and all identified surface cells (Right) (Yang and Gao, 2020).

As seen in Figure 8, only cells containing positions of surface particles are identified as surface cells. It is therefore likely that some parts of the isosurface will reside in cells not identified, in cases where a particles radius extends beyond its containing cell (see Figure 17). Consequently, it is still necessary to identify surface grid vertices for MC using bounding volumes for each surface particle. Ideally, and in contrast to this scheme, scalar field evaluation could be performed only at locations within the identified cells, removing the need to iterate over the surface particles constructing bounding volumes and performing intersection tests.

Nishidate and Fujishiro (2024) update this technique by introducing the use of a two-level uniform spatial grid and adjusting the surface cell identification logic. The two-level grid comprises a coarse grid of so-called *blocks* and a fine grid of *cells*. Mapping cells to blocks and vice versa can be done with simple index calculations, meaning no indirection table is needed. As per Yang and Gao (2020), cells track the number of particles they contain, however a chosen maximum is imposed. Each block likewise tracks the number of cells it contains that are influenced by particles.

Blocks containing surface cells and, subsequently, the surface cells contained by those blocks are identified. By only checking cells contained within the identified blocks, the workload is reduced. Blocks are identified as surface blocks when they are neither completely empty nor completely full of cells containing particles. They note that this scheme is a heuristic, and the technique wouldn’t account for small internal cavities or bubbles, however this is deemed

acceptable as only the outer surface is generally required for pleasing rendering results.

A cell is discarded if all 27 adjacent cells including itself are either entirely empty or fully filled with particles. Those remaining are surface cells. An illustration of the whole process can be seen in Figure 9.

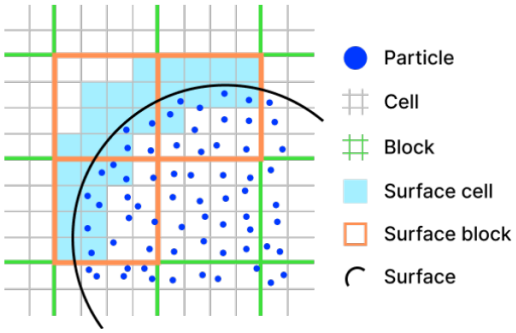

Figure 9 - Two-level grid structure, showing detected surface blocks and cells (Nishidate and Fujishiro, 2024).

As seen, an adequate buffer zone of surface cells is detected, allowing scalar field evaluation to simply be performed at locations within these surface cells. This scheme works for MC but also lends itself nicely to use of bricks of signed distance values, whereby however many required bricks can be constructed for each surface cell. Through use of this technique the researchers were able to render simulations consisting of hundreds of thousands of particles at real-time framerates, with timings appearing to increase logarithmically as the particle number increased.

2.4 Nearest Neighbour Search

Any given field value within an SDF is determined by the weighted average of surrounding particle positions and their neighbours within a fixed support radius h. A naïve approach to neighbour finding would involve calculating the distance between every particle and every other particle in the simulation. This is bad practice as visiting particles outside the support radius should be avoided where possible, to prevent redundant computation. Approaches which optimise the nearest neighbour search (NNS) are therefore highly desirable.

Binning particles into cells in a uniform spatial grid with cell side length <= h, means that only particles in the $3^3 = 27$ adjacent cells need be checked to see if they fall within the support radius, as shown in Figure 10.

Since each evaluated position in the SDF will therefore only need to access a small, spatially coherent subset of the particle data, it is desirable to access the particle data in a spatially coherent order to minimise cache misses, as shown in Figure 11.

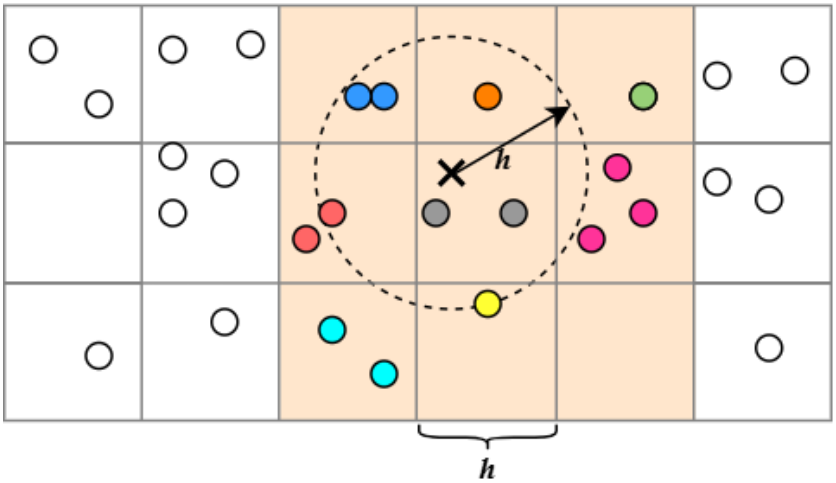

Figure 10 - Only 27 adjacent cells surrounding the evaluation position need be checked.

Figure 11 - Order of particle memory access. Random memory reads (Top) and spatially coherent reads (Bottom).

2.4.1 Particle Reordering

Green (2010) proposes using a radix sort to produce a list of particle indices sorted by cell, by sorting by the radix, or unique digits, of cell indices, as shown in Figure 12. Starting with the least significant digit, particle indices are grouped before the process cycles onto the next digit, until all digits in all indices have been processed.

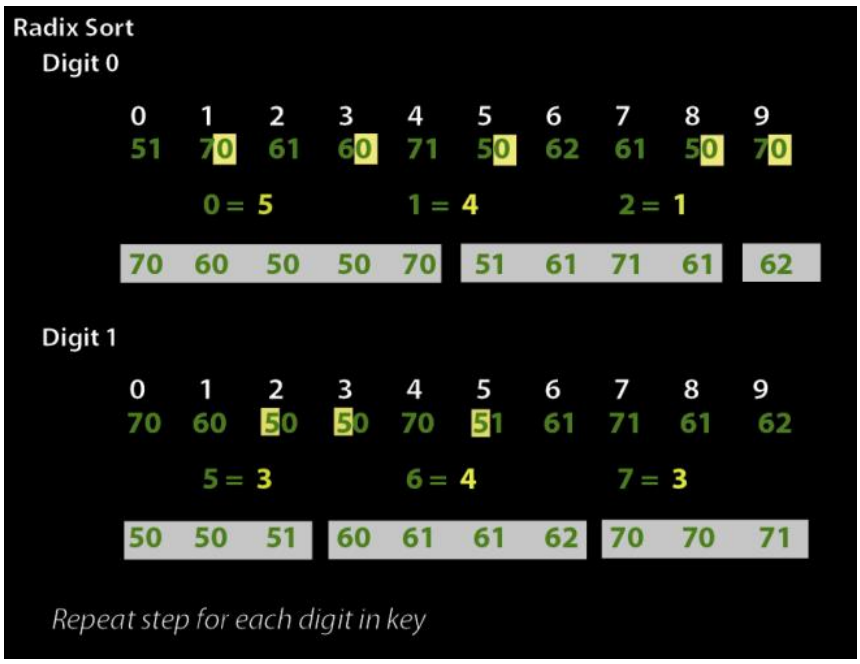

Figure 12 - Radix sort (Hoetzlein, 2014).

In order to make use of the sorted particle indices list, an array of addresses pointing to the start of each cell within said list is built.

Hoetzlein (2014) points out the radix sort performs unnecessary work as it uses multiple passes to preserve intra-cell particle order, yet the only goal is to group the particles by their cell, so the final position of each particle within each cell is irrelevant. The author therefore instead proposes a counting sort utilising a *prefix scan* of the number of particles in each cell. As seen in Figure 13, the number of particles in each cell is counted and placed in an array. Each counter is incremented atomically to obtain an intra-cell offset per particle. Inter-cell offsets are then found by scanning the particle counts array and writing the running total to each element in a new array (otherwise known as a prefix scan). The position of each particle index within the sorted list is found by combining the intra- and inter-cell offsets.

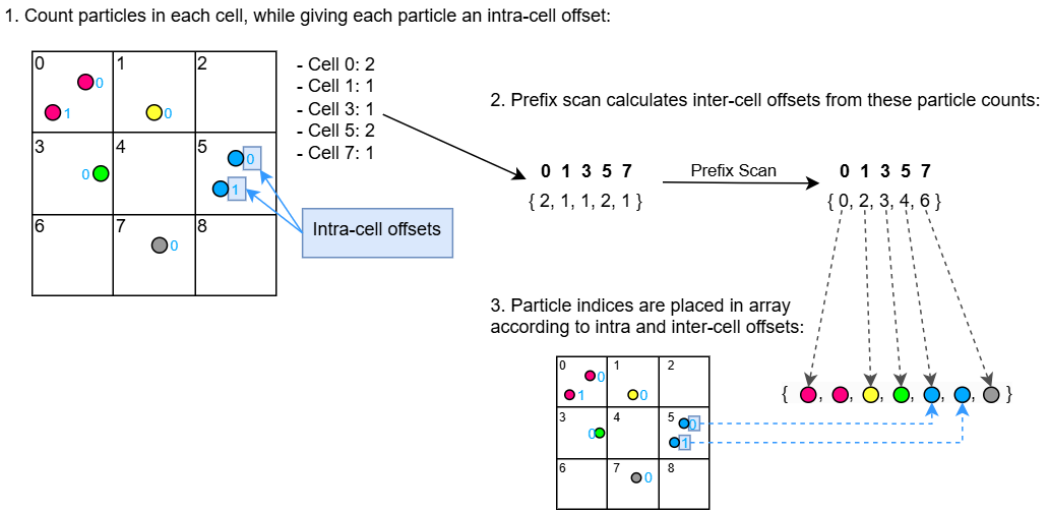

Figure 13 - Counting sort as described by Hoetzlein (2014).

Liu *et al*. (2023) follow the Hoetzlein (2014) method, improving upon it by utilising a more advanced *single-pass global prefix scan*, discussed below in Section 2.4.2. They also sort the full particle data, instead of just particle indices, to improve cache coherence when accessing the data of neighbouring particles.

The authors furthermore present a novel method for efficiently storing a neighbour list per particle representing each neighbour with a single bit, and a new algorithm for fast traversal of this list using clever bitmask operations. While impressive, this is used to accelerate processing of the actual fluid simulation, where multiple passes of NNS are required for various calculations; as SDF evaluation only requires a single NNS pass, this algorithm is not clearly applicable to the current research.

2.4.2 Prefix Scan

As explained above, the counting sort used by Hoetzlein (2014) and Liu *et al*. (2023) relies on a prefix scan which, given an input array of values, builds an output array containing the running total of the input list (not including the input element at the current index), as shown in Listing 2. This is specifically called an *exclusive* prefix scan.

Input:	{ 0, 1, 2, 3, 4, 5 }
Output:	{ 0, 0, 1, 3, 6, 10 }

Listing 2 - Example exclusive prefix scan input and output.

The obvious method is to sequentially iterate over the input list, writing a running total back to the output list. This has been parallelised in various ways.

One of the oldest and most widely used methods, including by Hoetzlein (2014), is a *3-pass reduce-then-scan* prefix scan presented by Blelloch (1993). The input array is treated as the leaf nodes of a balanced binary tree, and three kernels are required to perform the scan. Stepping from the leaf nodes to the root, an 'Up Sweep' reduces (sums) both child nodes at each step. A 'Down Sweep' then replaces the root with 0 (or some other specified identity value) and steps down through the tree, ‘scanning’. At each node, the value of the right child is set to the sum of the current node’s value and the left child’s value, and the current node’s value is copied to the left child. This process is shown in Figure 14.

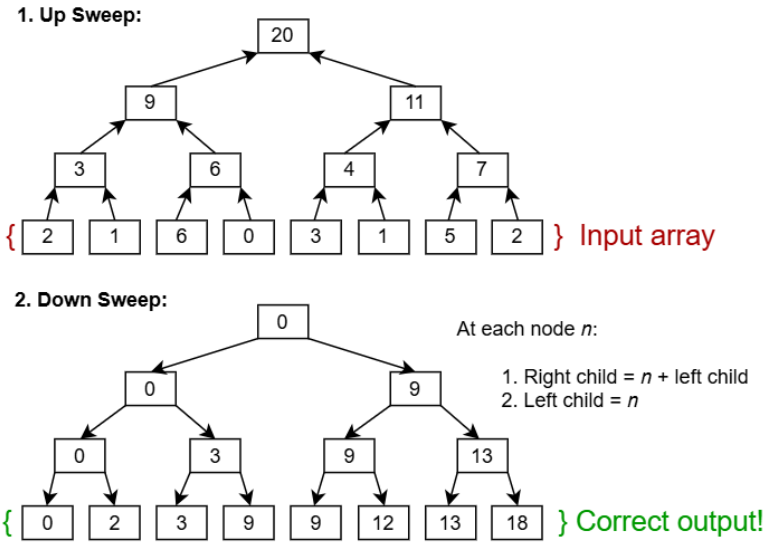

Figure 14 - Parallel prefix scan up sweep and down sweep.

To parallelise this for p processors, the input array is split into p subarrays, and each processor carries out the up sweep (reduction) locally on its subarray. An intermediary ‘root scan’ pass then performs both an up and down sweep to scan the list of subarray totals, which are used as identity offsets in the final, local down sweep pass.

One of the major improvements Liu *et al*. (2023) add to Hoetzlein’s (2014) method is to use a newer, *single-pass decoupled look-back* scan proposed by Merrill and Garland (2016). As before, each processor carries out a local reduction, however there is no root scan phase. Processors are ‘chained’; the 0th processor begins a local down sweep, while all other processors begin polling their predecessors. As

illustrated in Figure 15, this scan progressively checks further back along the chain, aggregating reductions until a prefix is found and added to the running aggregate, or until all previous reductions are aggregated. This final prefix value is used as the local identity in the down sweep.

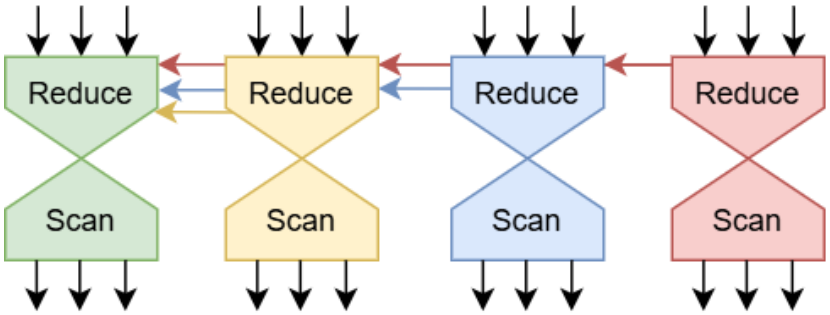

Figure 15 - Single-pass decoupled look-back scan.

Previous chained scans have each processor wait for its predecessor in the chain to signal completion. Such scans avoid redundant work but are bottlenecked by signal latency.

2.5 Summary

This chapter has examined various techniques involved in the rendering of particle-based fluid simulations. An SDF representing the fluid body is often evaluated and an isosurface is extracted to be rendered. Polygonal techniques generate an intermediary mesh representation conforming to the isosurface, to be rendered through rasterization. While fast, this is merely an approximation and cannot accurately capture details in the surface geometry without generating a dense mesh. Rendering can instead be achieved without polygonal representation, by directly ray tracing the isosurface through sphere tracing. To avoid re-evaluation of the SDF at every step along each ray, the SDF can be discretised and precomputed, storing values in a 3D texture to be sampled during ray tracing.

There have been various techniques proposed to reduce the latencies and memory usage of these methods. Nishidate and Fujishiro (2024) introduce a narrow band technique for minimising the computational complexity of scalar field evaluation by limiting the particles considered to a region of space close to the isosurface, using a two-level uniform spatial grid. Liu *et al.* (2023) propose a modern NNS algorithm ensuring cache coherency to further minimise SDF evaluation latency. Söderlund, Evans and Akenine-Möller (2022) find that SBS, a method in which sparse bricks of SDF values are stored and a BVH containing

their AABBs is constructed, offers a good compromise between low memory usage and fast ray traversal.

It is apparent that there are several ways in which the abovementioned techniques can complement each other effectively. Both the narrow band and NNS methods utilise uniform spatial grids; the fine grid of cells constructed for the narrow band method may be retained and repurposed for use in the NNS technique. Furthermore, a key operation present in both schemes is the counting of particles within each grid cell and the assignment of intra-cell indices to particles (in the case of the narrow band method, intra-cell indices are determined as the cell counter of the containing block is only incremented upon discovery of the first particle within a cell); this can thus be performed only once and the information shared between the schemes. Finally, cells containing the isosurface identified by the narrow band method can further be used to provide bounds in world space in which to construct the bricks and AABBs used by the SBS, a process which may utilise the NNS scheme to minimise SDF evaluation calculations.

It is speculated that consolidating these techniques into a unified pipeline will provide an effective means of reducing the latency and memory consumption of the non-polygonal rendering of particle-based fluid simulations through ray tracing.

Chapter 3 Methodology

Recognising the similarities shared between the narrow band technique of Nishidate and Fujishiro (2024), the NNS process set out by Liu *et al.* (2023) and the SBS method as per Söderlund, Evans and Akenine-Möller (2022) as explained at the end of the last chapter, an application was developed to implement a unified pipeline combining these techniques and evaluate its effectiveness at reducing the latency and memory usage of the non-polygonal rendering of SDF representations of particle-based fluid simulations through ray tracing, compared with simpler schemes.

The application was developed in C++ with Direct3D 12 (Microsoft, 2021), using DirectX Raytracing (DXR) (Microsoft, 2023) to leverage dedicated ray tracing hardware. For debugging and validation during development, RenderDoc (Karlsson, 2025) and the NVIDIA Nsight Graphics (NVIDIA, 2025b) Frame Debugger and Aftermath Monitor were used to inspect internal GPU activity and diagnose issues.

As making improvements to the underlying fluid simulation is not the focus of this investigation, and for ease of implementation, a true SPH simulation was not used to manipulate particle positions over time, electing instead to implement simple simulations approximating fluid behaviour.

Three pipelines were developed, henceforth referred to as the *naïve*, *simple* and *complex* pipelines; an overview is shown in Figure 16. The naïve pipeline re-evaluates the SDF at every sphere tracing step along each ray, performing calculations incorporating every particle in the simulation. The simple pipeline discretises and pre-calculates the SDF, storing a dense 3D volume of values to be sampled during sphere tracing. The complex pipeline utilises the abovementioned techniques to further minimise computational complexity and memory usage.

Unit testing was carried out to investigate performance metrics, memory usage and visual fidelity of the three pipelines under varying conditions and across multiple test systems.

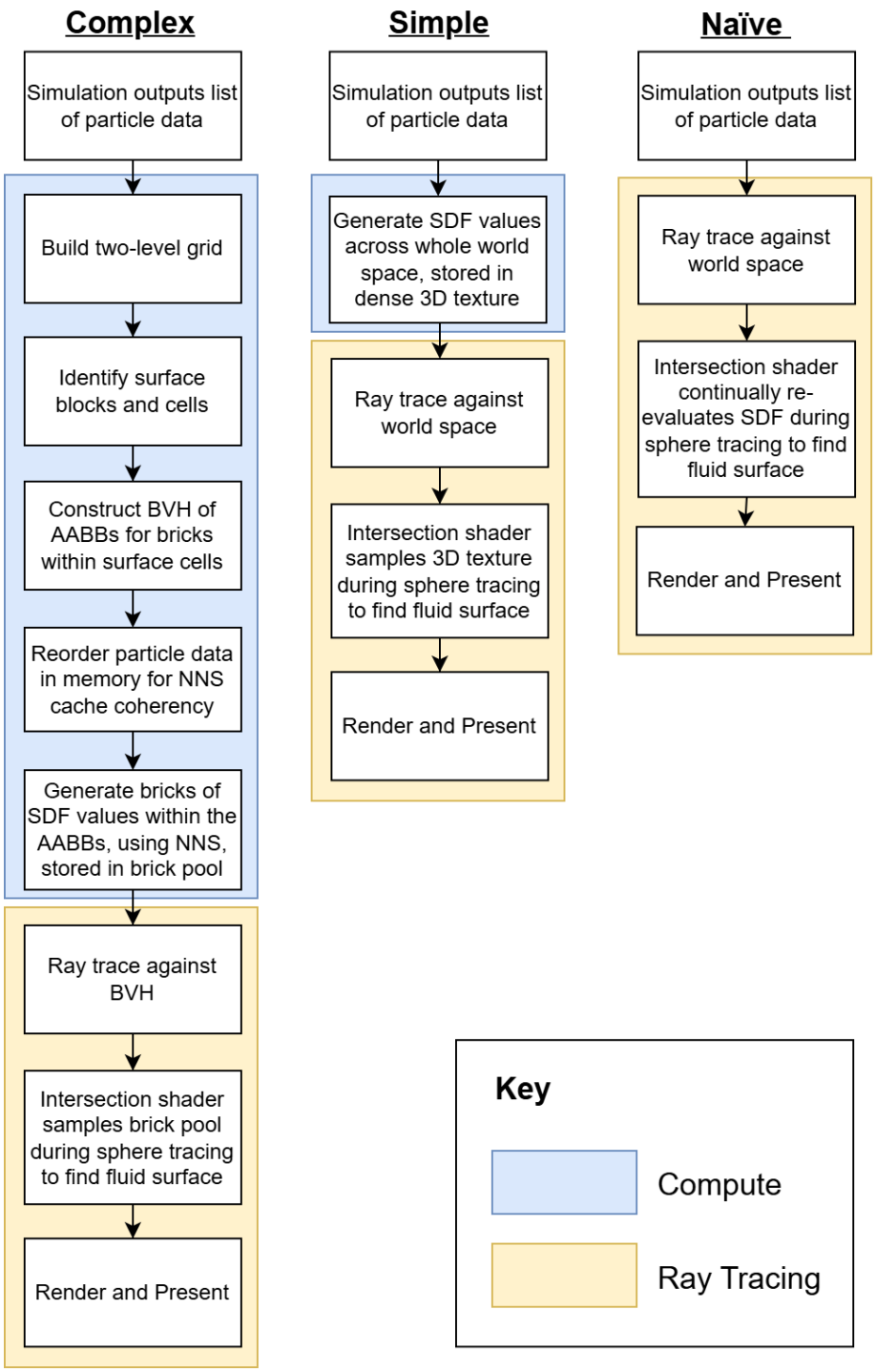

Figure 16 - Overview of the implemented pipelines.

The following sections break down the implementation details of all stages across all three pipelines, along with the testing methodology used.

3.1 Narrow Band Method

A method to limit SDF evaluation to the narrow band region around the fluid surface and thus reduce computational latency was implemented broadly

following the process described by Nishidate and Fujishiro (2024). While they apply this in the context of the Marching Cubes (MC) algorithm, this research applies it to SDF evaluation for non-polygonal surface reconstruction. This occurs in three main phases: grid construction, surface block detection and surface cell detection.

3.1.1 Grid Construction

The two-level grid structure is comprised of a 'coarse' grid of *blocks* and a 'fine' grid of *cells*. Each grid stores its respective data in a structured buffer, the elements of which are defined in Listing 3.

```
struct Cell
{
   uint particle_count_;
};

struct Block
{
   uint non_empty_cell_count_;
};
```

Listing 3 - Two-level grid data structure.

Cells track the number of particles they contain, whereas blocks hold the number of contained cells with one or more particles. Where a block contains no cells with particles but may still contain part of the isosurface, this value is incremented (see below for further explanation). The 'grid construction' stage simply consists of setting these values for each cell and block.

The grid is divided so that one block contains 64 cells in a 4^3 configuration. This arrangement maps well to the hardware as many consumer GPUs have a maximum of 64 resident warps per Streaming Multiprocessor (SM) (NVIDIA, 2025a, table 24) and is the suggestion of Nishidate and Fujishiro (2024).

Before the grid is constructed, a pre-processing kernel clears all counts from the previous frame. The grid construction process then takes place, the pseudocode for which is shown in Listing 4. This stage consists of filling in the counters for all cells and blocks. Enough threadgroups are dispatched so that one thread can run for each particle.

```
1. Calculate index of cell containing particle
2. Atomically increment cell's particle count and get particle's intra-cell index
3. If intra-cell index == 0:
    3.1. Build list of blocks adjacent to cell
    3.2. For each block in the list if non-empty cell count == 0:
        3.2.1. Atomically increment block's non-empty cell count
```

Listing 4 - Grid construction pseudocode.

Alongside step 2, the cell and intra-cell indices are stored in the particle's data, to be used later during the particle data reordering stage, discussed in Section 3.2.

In step 3, when a particle is the first in a cell, the counter of the containing block is incremented, and crucially up to 7 neighbouring blocks may also need their counter incremented. This occurs in situations where the particle's radius extends beyond the containing block, as seen in Figure 17.

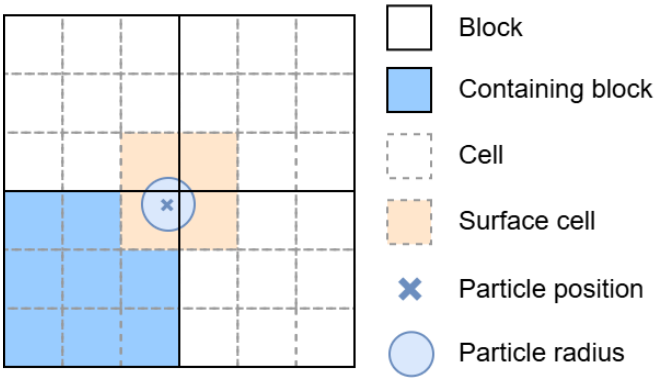

Figure 17 - Particle radius extends beyond the containing block.

As seen in the 2D example, up to 3 adjacent cells outside of the current block may contain the isosurface. Therefore, the 3 adjacent blocks containing these cells must have their non-empty cell counts incremented if they are empty, so that the relevant cells may later be identified as surface cell, as discussed in Section 3.1.3. Which blocks this applies to is determined by the position, within the containing block, of the cell containing the particle. Extending to 3D gives up to 7 neighbouring blocks.

3.1.2 Surface Block Detection

Following grid construction, surface blocks and cells are identified. Arrays of integers, one for surface blocks and one for surface cells, store the indices of those identified. A struct, shown in Listing 5, is filled out to track the count of each.

```
struct GridSurfaceCounts
{
    unsigned int surface_blocks;
    unsigned int surface_cells;
};
```

Listing 5 - Struct for tracking counts of surface cells and blocks.

A thread is launched per block to identify surface blocks; blocks are identified as such when they are neither completely empty nor full (0 < non-empty cell count < 64). Full blocks at the edge of the grid are also identified, so that the surface is rendered along the boundary. Pseudocode for this stage is shown in Listing 6.

```
1. Use thread dispatch ID as block index
2. If the block's non-empty cell count == 0 or 64:
   2.1. Return
3. Atomically increment surface block count and
   store the index.
```

Listing 6 - Surface block detection pseudocode.

The index of a block is a unique value representing the flattening of the block's (i, j, k) coarse grid coordinates, and the same logic applies to cell indices.

3.1.3 Surface Cell Detection

Before the surface cell detection stage, the total number of surface blocks is read back to the CPU so that one threadgroup can be dispatched for each surface block. The threads in a threadgroup are arranged in a 4^3 configuration to match the layout of cells per block described in Section 3.1.1, so that one thread is run for each internal cell of each surface block. The pseudocode for this stage is shown in Listing 7.

A cell is not at the surface when all 27 neighbouring cells including itself are the same *fullness*, meaning completely empty or completely full. A cell is empty when its particle count is 0, and full when the particle count is equal or greater to a chosen threshold. This max particle count value was set to 8, in line with Nishidate and Fujishiro (2024).

1. Get block index from surface block index array using threadgroup ID
2. Calculate cell index using block index and offset from threads 3D index within thread group
3. If cell's particle count > 0 or < max particle count:
    3.1. Increment the count of surface cells and store the index.
    3.2. Return
4. For each adjacent cell:
    4.1. If adjacent cell's fullness is different to current cell:
        4.1.1. Increment the count of surface cells and store the current cell's index.

Listing 7 - Surface cell detection pseudocode.

In step 1, groupshared memory is exploited to minimise global memory accesses by assigning block index retrieval exclusively to the leading thread in the group.

In step 3, a cell is identified as a surface cell if it is neither empty nor full, as clearly the 27 neighbouring cells are not all completely empty or full. If the cell is completely empty or full, then all remaining adjacent cells are checked until one is found to have a different fullness. For example, if the current cell is empty and the adjacent cell contains particles.

In steps 3.1 and 4.1.1, surface cell indices are stored to be used later during the AABB construction and brick pool evaluation stages, discussed in Section 3.3.2.

3.2 Nearest Neighbour Search

As explained in Section 2.4, an efficient nearest neighbour search (NNS) algorithm can expedite the process of locating particles within a fixed radius h of a position, necessary during scalar field evaluation (see Section 2.1.1), by grouping particles into cells in a uniform spatial grid with cell side length <= h. Hence, only particles in the $3^3 = 27$ adjacent cells surrounding the position being queried need to be checked. This synergises well with the narrow band method described in Section 3.1, as the 'fine' grid of cells constructed during that process may be reused for the NNS purpose.

The method set out by Liu *et al.* (2023) was followed. Recall from Figure 13 that the list of counts of the number of particles contained by each cell is required to be processed by a prefix scan. Adding further to the synergy mentioned

above, this list of particle counts is conveniently already available as it is constructed as a key component of the narrow band system, as seen in Listing 3. Furthermore, to ensure thread safety the particle counting is performed using an atomic add, which additionally returns the original value of the counter, which can be used as the particle's intra-cell offset required by the sorting stage. Each particle stores the index of the cell containing it and its own intra-cell offset.

3.2.1 Particle Reordering

As per Liu *et al*. (2023), the *single-pass decoupled look-back* scan introduced by Merrill and Garland (2016) (see Section 2.4.2) was utilised to calculate the inter-cell offsets used during the reordering of particle data. This is a complex algorithm, the details of which are not necessarily the focus of the current research; thus, a readily available, open-source implementation (Smith, 2024) was selected for use due to its ease of integration and popularity.

The structured buffer of particle counts per cell is provided to the prefix scan, and a second buffer of the same size is filled with the output, consistent with Listing 2. Accessing this output buffer using a cell's index returns the inter-cell offset necessary for particles contained by the cell.

Liu *et al*. (2023) recommend that in order to improve data locality for cache coherency during NNS, the data associated with every particle within a cell should be placed contiguously in memory – not just indices in a lookup table. To facilitate this, a particle reordering kernel launches a thread for every particle to copy data from the original unsorted buffer to a secondary sorted buffer, as illustrated in Listing 8. The use of two buffers for holding particle data does mean twice the memory usage but was chosen for ease of implementation. The results presented in Chapter 4 will show that cache hit rate is indeed improved, thus this is a compromise worth making.

1. Get particle data from unordered buffer, using thread dispatch ID
2. Get cell index and intra-cell offset from particle data
3. Get inter-cell offset from scan output buffer, using cell index
4. Set new index = intra-cell offset + intra-cell offset
5. Copy particle data to ordered particle buffer, using new index

Listing 8 - Particle reordering pseudocode.

3.2.2 Search Algorithm

The algorithm for finding particles within fixed radius h of a position by checking the 27 adjacent cells during SDF evaluation is straightforward. As will be explained in Section 3.3.2, a list of 27 cell indices is built beforehand. Pseudocode for the search algorithm is presented in Listing 9.

```
1. For each cell index:
    1.1. Ensure cell index is valid
    1.2. Get cell particle count from particle counts
         buffer, using cell index
    1.3. Get inter-cell offset from scan output
         buffer, using cell index
    1.4. For i = 0; i < particle count; i++:
        1.4.1. Particle index = inter-cell offset + i
        1.4.2. Get particle data from ordered
               particle buffer, using particle index
        1.4.3. Check if particle is within radius of
               position
```

Listing 9 - NNS search pseudocode.

In step 1.1, a cell index is classed as valid when it is greater than -1 and less than the total number of cells.

3.3 Scalar Field Evaluation, Discretisation and Rendering

As indicated at the beginning of this chapter, three differing pipelines were implemented to be compared with one another; the ‘naïve’ pipeline, which performs many iterations of calculations per ray, the ‘simple’ pipeline which precomputes these calculations once in parallel, and the ‘complex’ pipeline utilising the narrow band and NNS methods described thus far to further minimise computational complexity.

3.3.1 SDF Evaluation

As explained in Section 2.1.1, to evaluate an SDF at a given position in space, the signed distances to surrounding particles are combined and smoothed using a function which blends the signed distance values over a fixed radius of influence h.

To implement this, an initial distance value is set to an arbitrarily high magnitude. The algorithm then iterates over all relevant particles and combines

their signed distances with the current distance value using the smooth minimum function used in *Dreams* (Evans, 2015), defined in . Pseudocode for the process is shown in Listing 10.

Given position in space $\boldsymbol{x}$,

1. Set distance = 1000
2. For each particle:
   2.1. Set distance1 = $\phi(\boldsymbol{x})$
   2.2. Set distance = SmoothMin(distance, distance1)
3. Return distance

Listing 10 - SDF Evaluation pseudocode.

In step 2.1, the signed distance ϕ to a particle is as defined in . The naïve implementation iterates over all particles in the simulation, blending all distance values regardless of how far away each particle is from the evaluation position $\boldsymbol{x}$. As explained at the beginning of this chapter, the complex implementation instead utilises the method of nearest neighbour search (NNS) described in Section 3.2 combined with the narrow band method described in Section 3.1 to minimise the number of contributing particles.

As the NNS algorithm locates particles within a fixed radius h_{nns} <= the side length of cells in the grid (as per Figure 10), the radius of influence h_{in} considered by the smooth minimum function for each particle cannot be greater than h_{nns}. The current implementation sets both h_{nns} and h_{in} equal to the side length of the cells. The radius r_i of each particle i is set to half this length. As the signed distance to a particle already accounts for r_i, the value of h used by the SmoothMin function as defined in is in fact equal to r_i (since $h_{in} = 2r_i$).

3.3.2 Discretisation

As explained in Section 2.2, the naïve implementation performs SDF evaluation at every raymarching step of the sphere tracing process, which has a multilinear time complexity of O(mn) where n is the number of particles visited at each iteration and m is the number of sphere tracing iterations. The time complexity can be reduced to O($m + n$) by discretising and precomputing the SDF, storing values computed in parallel at regular intervals in space into a 3D texture to be sampled with constant time complexity in each sphere tracing iteration.

The simple implementation stores a dense, fixed-size 3D volume of SDF values as *Claybook* (Aaltonen, 2018) does, however only makes use of a single mip level. To generate this, a thread for each voxel according to the specified texture resolution is launched, in groups of 32^2 threads. The position of the voxel within the 3D texture is mapped to a corresponding location in world space and the SDF is evaluated at that location, using the naïve method described in Section 3.3.1, blending the signed distances to all particles in the simulation. The returned value is written to the texture, and the result is a dense volume of SDF values, an example of which can be seen in Figure 18. Both this texture and the brick pool described later store values as 16-bit signed normalised floats.

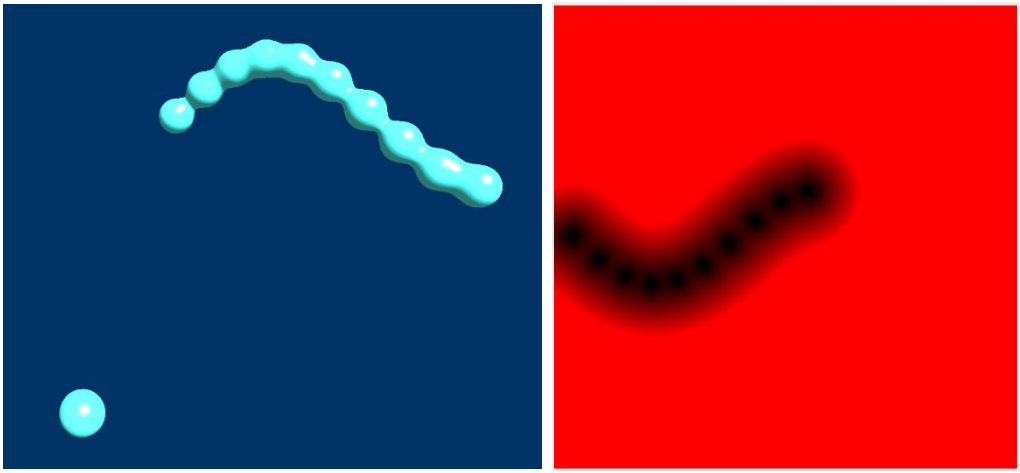

Figure 18 - A rendered scene (Left), and a cross section of the texture representing it, with a resolution of 256^3 (Right).

The complex pipeline instead discretizes the SDF similarly to the sparse brick set (SBS) method investigated by Söderlund, Evans and Akenine-Möller (2022). As stated in Section 2.2, SBS offers a good compromise between low memory usage (as is necessary in memory constrained scenarios such as fluid simulation) and fast ray traversal. Furthermore, the surface cells of the spatial grid identified by the narrow band process described in Section 3.1 offer convenient bounds within which to construct the bricks.

In the current implementation, a brick is a 10^3 grouping of voxels. Each brick is constructed to represent an area of world space bounded by an AABB and is placed compactly within a 3D texture known as the *brick pool* (the full process is described later in this section). While the brick pool is itself a dense 3D volume of SDF values, the bricks are mapped sparsely across the world space bounds, only representing areas of space located close to the fluid surface as identified by the narrow band process (as mentioned above). A visualisation of this is shown in Figure 19. The result is that less voxels are required overall to represent a scene, as many fewer voxels corresponding to empty space are evaluated, unlike in the simpler dense volume method.

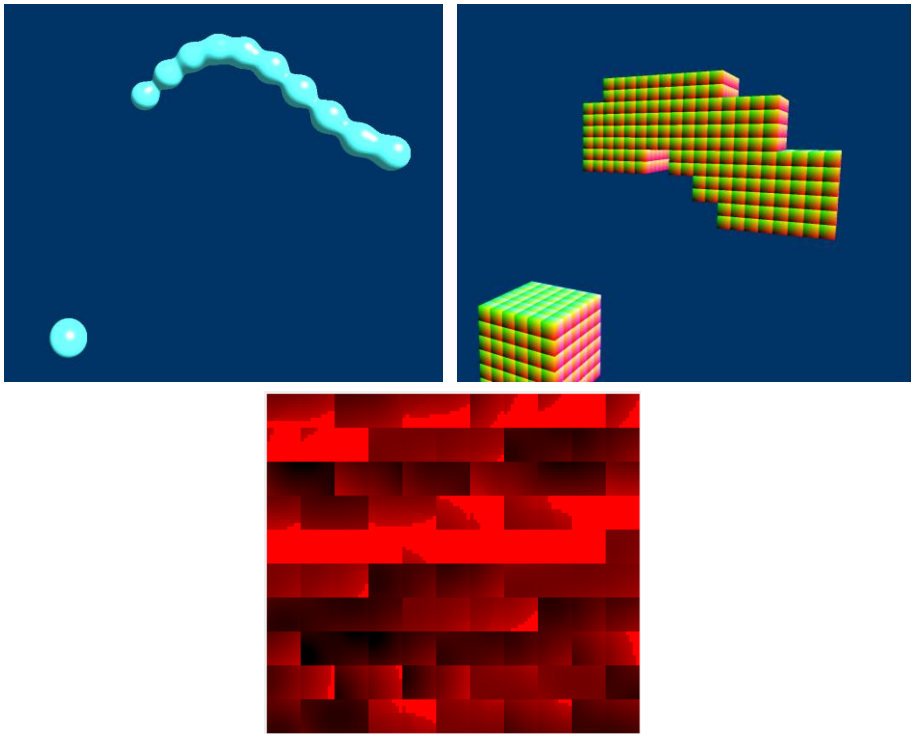

Figure 19 - A scene (Left), visualised bounding boxes for the constructed bricks (Right) and a cross section of the brick pool (Bottom).

During rendering, hardware sampling with bilinear filtering is utilised to efficiently smoothen the discretised surface. Bilinear filtering gathers and interpolates the values from a 2^3 neighbourhood of voxels surrounding the sample position within the texture. Due to this, the outermost layer of voxels in a brick are used purely to store adjacency data; without this, if two bricks adjacent in the brick pool are not adjacent in world space, sampling near the edge they share within the brick pool would result in incorrect visual results.

Each brick is 10^3 voxels, so only the inner 8^3 voxels, called the *core voxels*, are sphere traced during rendering. The 10^3 resolution was chosen as $10^3 = 1000$ is

the maximum cubic number under the thread group size limit of 1024. Maximising the brick dimensions minimises the percentage of voxels per brick used for adjacency data; for an 8^3 brick such as those used by *Dreams* (Evans, 2015), 58% of the brick stores adjacency data, whereas for a 10^3 brick this drops to 49%.

Before the brick pool can be constructed, the number required, and the necessary AABBs must first be ascertained. Enough bricks are generated to fill each surface cell identified in the narrow band process. The number of bricks which fit into a cell is calculated by stipulating that voxels are the same size in world space as they would be when using the simple dense 3D texture method. Under the current implementation there are 16 cells per axis in the grid, and 8 core voxels per axis within each cell, so the bricks per axis per cell is calculated as:

$$Bricks\ per\ axis\ per\ cell = \frac{Texture\ resolution}{16 \times 8} \quad (6)$$

Texture resolution is assumed to be cubic and is specified by its resolution in one dimension, thus for a specified texture resolution of 256^3 this comes out to 256/128 = 2, resulting in $2^3 = 8$ bricks for every cell. The total number of bricks is therefore 8 multiplied by the number of identified surface cells, which is read back from the GPU to the CPU after the surface cell detection stage. If the required number has changed since the last frame, the AABB buffer is reallocated.

AABBs are then constructed for each brick. An AABB is defined by minimum and maximum bounds as in Listing 11 and illustrated in Figure 20.

```
struct AABB
{
   XMFLOAT3 min_;
   XMFLOAT3 max_;
};
```

Listing 11 - AABB structure.

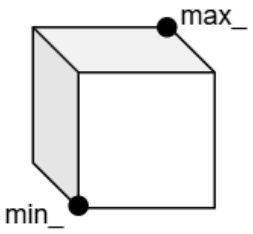

Figure 20 - AABB illustration.

Enough threads for the total required number of bricks are dispatched, and through simple calculations using the thread dispatch ID and the world space position of each surface cell (calculated by mapping the cell's grid space coordinates to world space), the bounds for each brick are found and stored in a structured buffer.

Following the AABB construction stage, the brick pool is generated. To allocate memory for the 3D texture, the CPU finds optimal dimensions for storing the

required number of bricks. Texture reallocation occurs whenever the required number of bricks exceeds the current maximum number which can be stored by the brick pool. Therefore, the brick pool will grow over time as more bricks are required, which is a compromise between minimising memory usage and the need to reallocate memory every frame.

A thread group per brick is dispatched with a group size matching the 10^3 configuration of a brick. A general outline of the process is shown in Listing 12.

```
1. Get aabb from aabb buffer, using thread dispatch ID
2. Get cell index from surface cell indices buffer
3. Build list of indices of 27 adjacent cells
4. Find position in world space using thread position
   within thread group and aabb.min_
5. Find position within brick pool to store SDF value
6. Calculate and store SDF value
```

Listing 12 - Brick pool construction pseudocode.

In step 3, a list of the 27 adjacent cells including the cell containing the current brick is built for use in the NNS algorithm. As all the threads in a thread group represent a single brick, groupshared memory (GSM) is utilised to expedite the process, allowing 27 threads to offset the cell index in parallel and share this with the group, rather than all threads performing 27 calculations serially. GSM is additionally employed to reduce global memory accesses by designating only the first thread within the group to retrieve the AABB and cell index.

The location of the current voxel within the brick pool is calculated based on the brick pool's dimensions, the brick index and the thread's index within the thread group.

Finally, the SDF is evaluated at the position in world space calculated in step 4, employing the NNS algorithm described in Section 3.2.2, using the neighbour list found in step 3. A closer look at the brick pool is provided in Figure 21.

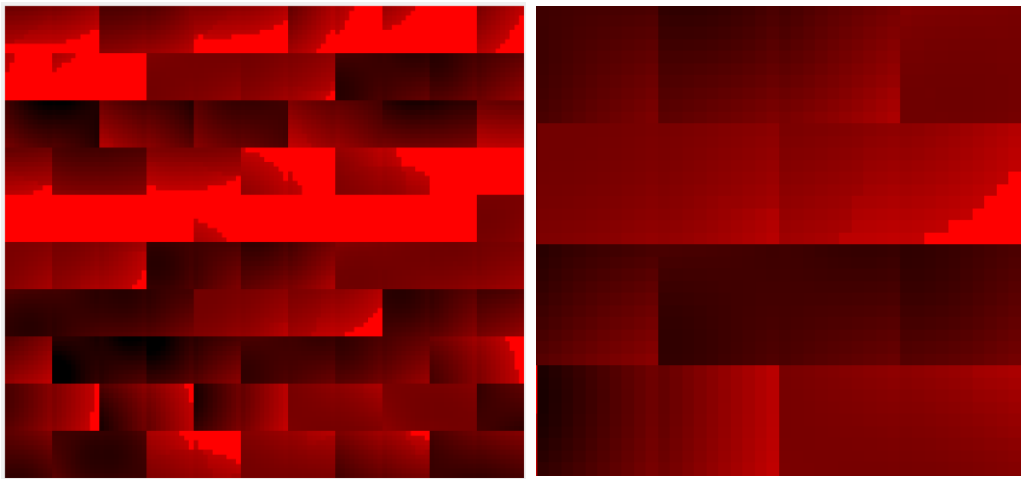

Figure 21 – Example brick pool cross section (Left) and an enlarged portion (Right).

3.3.3 Rendering

As per Söderlund, Evans and Akenine-Möller (2022), the implemented SBS method leverages DirectX Raytracing (DXR) hardware to accelerate ray traversal of a bounding volume hierarchy (BVH). This is a tree structure of AABBs used to rapidly locate the contained geometry, a possible example of which is illustrated in Figure 22.

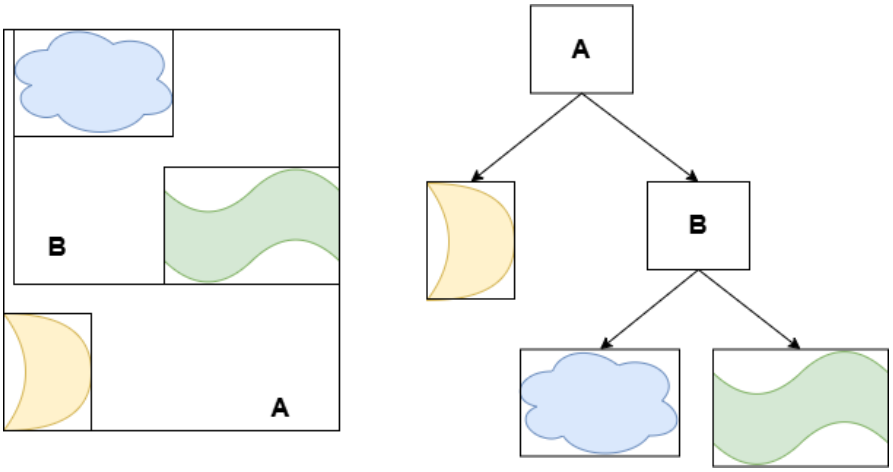

Figure 22 - BVH, shown in world space (Left) and the underlying tree structure (Right).

To construct the BVH, only the AABBs of the bricks are supplied to DXR to be used as the leaf nodes. Once constructed, the original AABB buffer is no longer required by DXR, however is retained in memory for use during a custom intersection shader.

Every frame, the acceleration structure is updated to reflect changes in the geometry, either modifying the AABB values, or reallocating the memory and rebuilding when the number of bricks changes.

In the case of the naïve and simple pipelines, a single AABB bounding the whole scene is used to construct the BVH at the beginning of execution and no further updates are made.

To perform ray tracing, for each pixel a ray generation shader uses an inverse view projection matrix to project the pixel's 2D screen space coordinate into a ray in world space, defined by an origin at the viewpoint and a direction vector, as represented in Figure 23. The final colour of each pixel is determined by shaders acting on each corresponding ray, as is also shown in the figure.

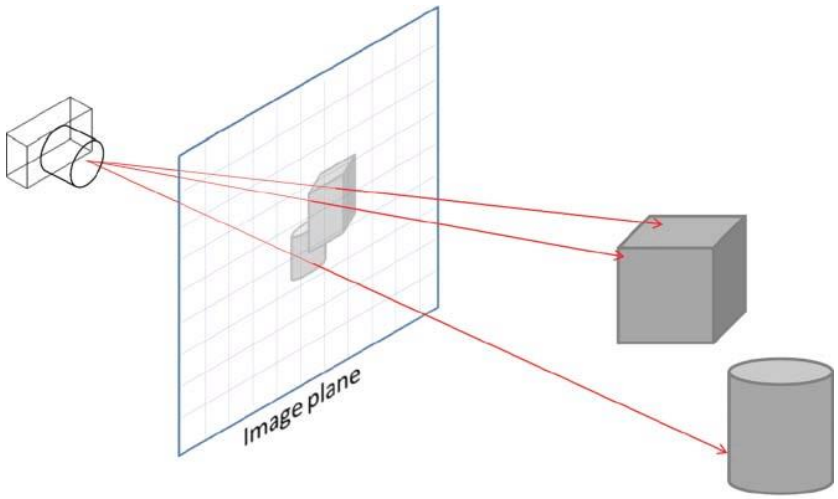

Figure 23 - General raytracing overview (Zhang, 2011).

The rays are cast against the scene, traversing the BVH to detect intersections with the supplied AABBs. Once a ray-AABB intersection is detected by the hardware, a custom intersection shader is triggered.

The only information provided by DXR regarding a ray-AABB intersection is the index of the AABB within the buffer of AABBs used to construct the BVH. Therefore, the intersection shader must fetch the AABB from the buffer and re-perform the intersection test. A ray R, with origin $\boldsymbol{o}$ and direction vector $\boldsymbol{d}$ is defined as:

$$\mathrm{R} = \boldsymbol{o} + t\boldsymbol{d} \tag{7}$$

The intersection test returns t_{min} and t_{max} values which bound the ray to the interval spanning the AABB as shown in Figure 24.

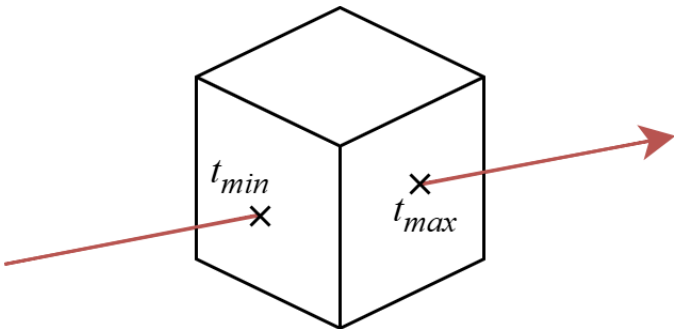

Figure 24 - Ray-AABB intersection. t_{min} and t_{max} values bound the ray to the points of intersection as shown.

The sphere tracing algorithm presented in Section 2.1.3 is employed to ray-march over the interval, checking for intersection with the SDF isosurface. The naïve implementation evaluates the SDF value for the current position in world space at each step. The simple and complex pipelines convert the position to the AABB's local coordinate system ranging from (0, 0, 0) to (1, 1, 1), which is used to sample the correct position within the 3D texture.

Once the point of intersection with the isosurface is detected, BVH traversal is terminated and a surface normal calculated through central differencing, adapting an implementation by Quilez (2015). This technique samples the SDF at 6 evenly distributed positions surrounding the point of interest calculates the difference between them to estimate a derivative. The naïve pipeline uses an arbitrarily small step size in world space, whereas the dense 3D volume is sampled at neighbouring voxels. The step size for the brick pool is limited to half a voxel to avoid linear sampling across brick boundaries as shown in Figure 25.

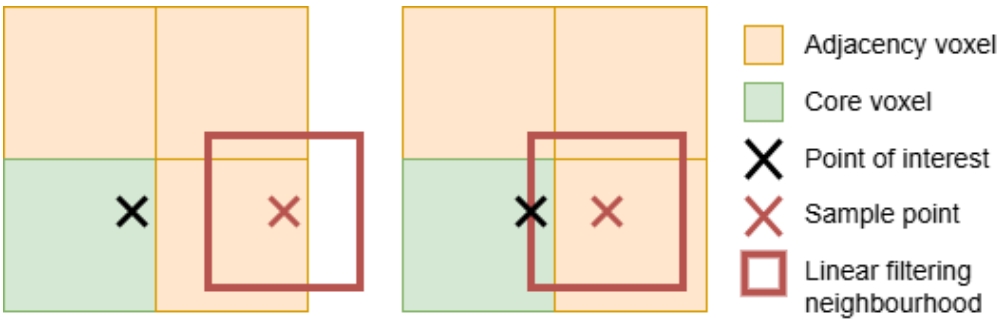

Figure 25 - Central differencing with step size = 1 voxel (Left) and 0.5 voxels (Right). The second case prevents sampling across brick boundaries.

3.4 Testing and Evaluation

Testing was conducted to assess the effectiveness of the complex pipeline over the less optimised methods, with an emphasis on performance metrics, memory usage and visual fidelity.

The systems used to run the tests had the following specs:

#	GPU	CPU
1.	NVIDIA RTX 4060 8GB, driver 572.60	AMD Ryzen 7 9800X3D
2.	NVIDIA RTX 4090 24GB, driver 572.16	AMD Ryzen 9 9950X
3.	NVIDIA GTX 1660 Ti Mobile 6GB, driver 576.28	Intel i7-9750H

Table 1 - Test systems used.

The majority of tests were performed on system #1, with only a few tests on the others to gain a sense of performance across differing hardware.

3.4.1 Performance and Memory Usage

The application was instrumented with the NVIDIA Nsight Perf SDK 2025.1 (NVIDIA, 2025c) to collect performance metrics across specified GPU workloads of interest over a given frame, referred to as ranges. The metrics collected, alongside their respective purposes, are presented in the table below. Throughput is the percentage of maximum attainable data processing rate achieved by a hardware unit. The focus of the testing and evaluation was on time duration, with most of the additional metrics only collected in case they prove helpful to aid in discussion of the observations made.

Metric	Purpose
Time duration (ns)	To compare latency of operations.
Peak SM throughput (%)	To measure peak utilisation of the busiest SM.
Warp activity (%)	To measure occupancy, or the extent to which the GPU is filled with actively scheduled warps.
L2 cache hit rate (%)	To assess efficiency of cache memory accesses.
RT core throughput (%)	To measure ray tracing hardware utilisation.
Warp launch stall reason (%)	To determine the most common barriers to warps being launched.
ALU and LSU throughput (%)	To determine whether workloads exhibit a greater demand for arithmetic operations or memory access.

Table 2 - GPU performance metrics collected for evaluation.

In addition to GPU performance metrics, the memory overhead incurred by resources in use was also measured. The targeted resources comprised all those that are not shared across the three different pipelines, including the buffers in use by the narrow band system (cells, blocks, surface cell indices, surface block indices, surface counts and counts readback buffers), simple dense 3D texture, the brick pool, AABB buffer, bottom-level acceleration structure (BLAS), ordered particle buffer, inter-cell index offsets buffer and buffers used internally by the prefix scan. The BLAS is the portion of the BVH generated by DXR containing the supplied AABBs.

Nsight Perf does not provide a mechanism for memory usage collection, so memory usage of a resource was calculated by the CPU upon allocation. For the majority of resources, which maintain a fixed size throughout execution, allocated memory is accordingly only calculated once, whereas for dynamic resources (the brick pool, AABB buffer and BLAS), this must be recalculated each time the resource undergoes a change in size. In the case of structured buffer resources, the size in bytes is calculated as the number of elements multiplied by the size of an individual element. For the texture and BLAS resources, a DirectX 12 device function, GetCopyableFootprints(), is supplied with a description of the resource and the size in bytes is output.

Three test scenes were developed to measure performance and memory usage under different conditions, by changing the way particles are generated and updated, shown in figure. The scene *Random* generates particles randomly throughout the world space and updates their positions by moving them back and forth along a random direction vector. This scene represents an abstract and unpredictable situation with particles fairly evenly distributed across the world space bounds. The scene *Grid* generates particles in a uniform grid configuration, ensuring that adjacent particles are in contact with one another, and particles remain stationary. The *Wave* scene approximates a more realistic water scenario, generating a compact grid of particles which are oscillated according to a sine function to give the appearance of a wave. Each particle has a 5% chance to be assigned a speed higher than the rest, to imitate small splashes in the fluid. All scenes have world space bounds ranging from (0, 0, 0) to (1, 1, 1).

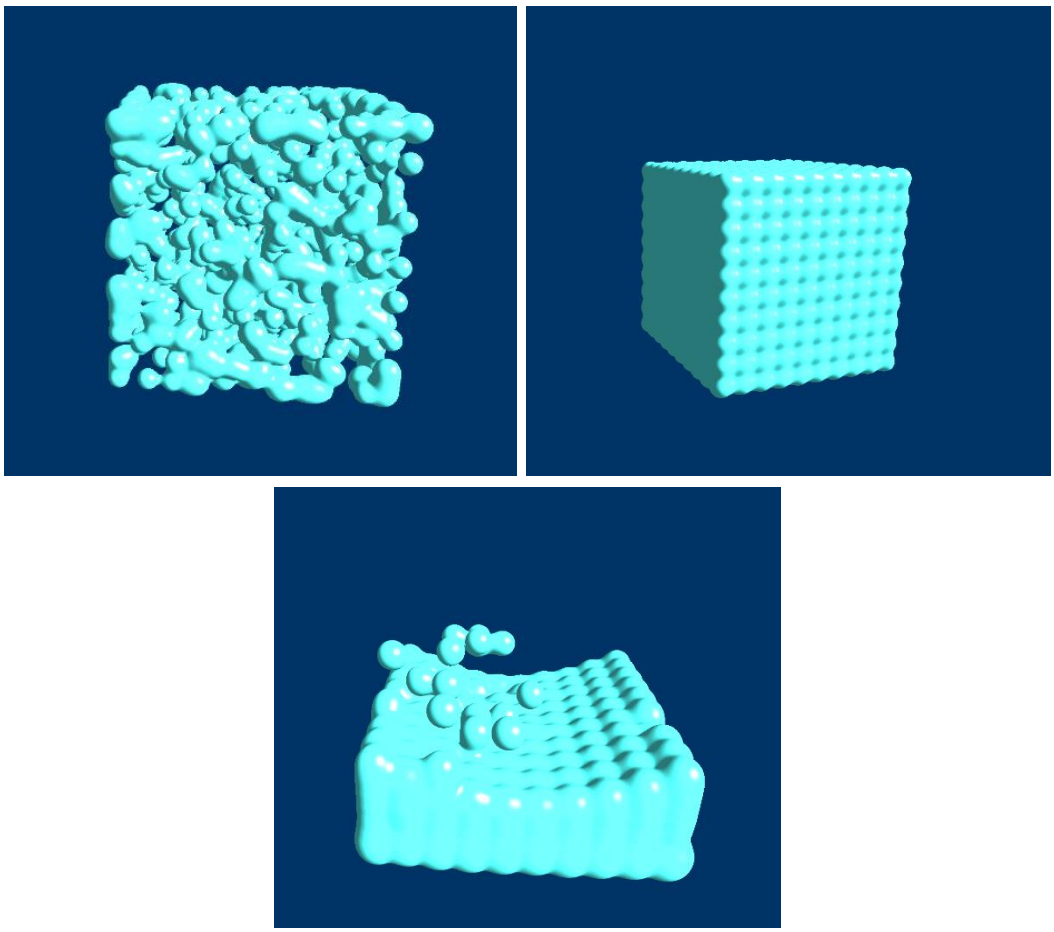

Figure 26 - Test scenes, with 1000 particles. Random (Left), Grid (Middle) and Wave (Bottom).

The application was configured to allow for the customization of several variables through command line arguments. Automated unit testing through use of a batch file was conducted to examine the impact of varying each individual variable, with default values assigned to those variables not under examination in a given test run. The table below shows these variables, the ranges of values tested, and the chosen default values. The default configuration was chosen such that it performs reasonably well on the main test system using the naïve pipeline, allowing room for more intensive settings to be tested.

Variable	Values Tested
Number of particles	1, 27, **343**, 1000, 10648
Texture resolution	128^3, $\mathbf{256^3}$, 768^3
Screen resolution	1280×720, **1920×1080**, 2560×1440, 3840×2160
View distance	1, **1.5**, 2, 2.5
Scene	Random, Grid, **Wave**

Table 3 - Variables and the ranges of values tested. Default values are indicated in bold.

To perform the tests, each configuration was ran utilising all three pipelines (naïve, simple and complex). In each test, 20 captures were taken at 0.5 second intervals, employing an orbital viewpoint to remove any view-dependent bias. Average values were output to CSV files for further refinement and examination in Microsoft Excel (Microsoft, 2025).

3.4.2 Visual Fidelity

Even if the complex pipeline offers tremendous improvements to rendering latency over the naïve and simple pipelines, it is likely not worth utilising if the visual fidelity is degraded to an unsatisfying level, as this risks disrupting player immersion. For the purposes of this study, visual fidelity will be defined as a measure of how strongly the fluid geometry and surface normals resemble that of the naïve implementation.

To assess visual fidelity, a fourth test scene, shown in Figure 27, was created with a static viewpoint as seen. The positions of the particles were chosen to allow varied normals across the fluid surface. The scene was rendered with a screen resolution of 2560×1440 employing each of the implemented pipelines, shaded to visualise the surface normals. The differences between captures were examined in Photoshop (Adobe, 2025).

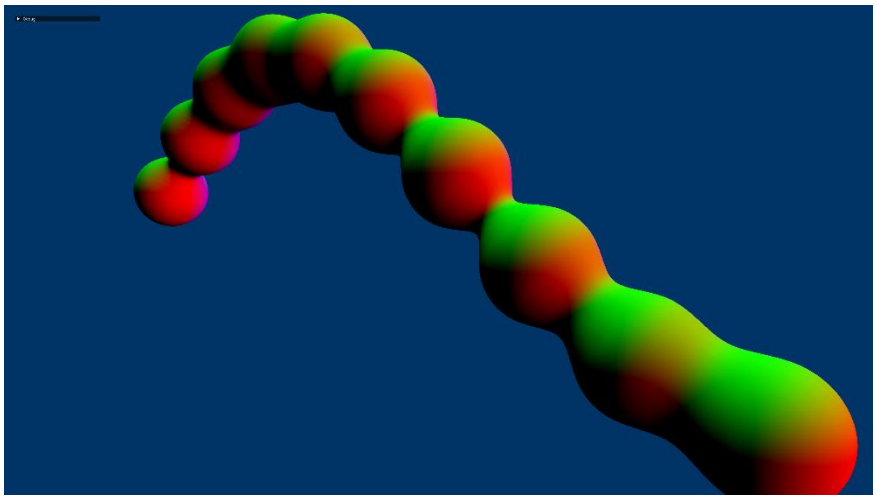

Figure 27 – Scene used for testing visual fidelity.

Chapter 4 Results

This chapter presents the results from the testing as described in Section 3.4. The full recorded data can be found in Appendix A. The following sections present in turn the findings regarding each of the variables tested as shown in Table 3, followed by the results of the visual fidelity testing.

Firstly, Figure 28 and Figure 29 show the average L2 cache hit rate and SM throughputs during SDF evaluation for all three pipelines. This data was taken from tests with the default variable configuration, but the trends were found to be similar across the majority of tests.

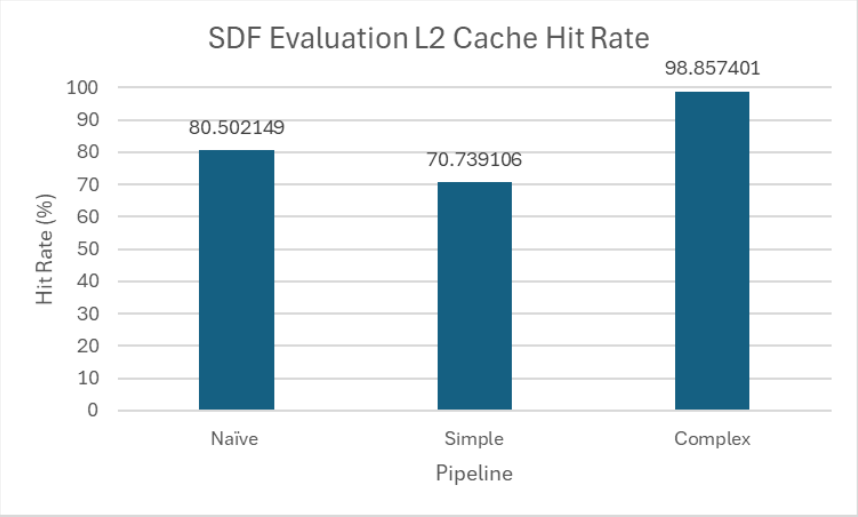

Figure 28 – Average L2 cache hit rate during SDF evaluation across pipelines with the default configuration.

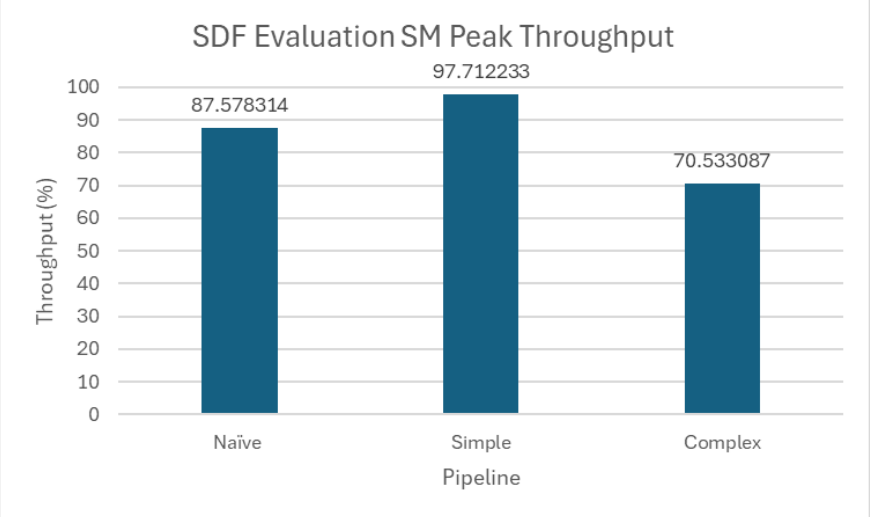

Figure 29 – Average peak SM throughput during SDF evaluation across pipelines with the default configuration

4.1 Number of Particles

Figure 30 shows the average frame times for each pipeline as the number of particles increases. For clarity, Figure 31 shows the results with the axes scaled logarithmically, and Figure 32 shows only the timings for the complex pipeline.

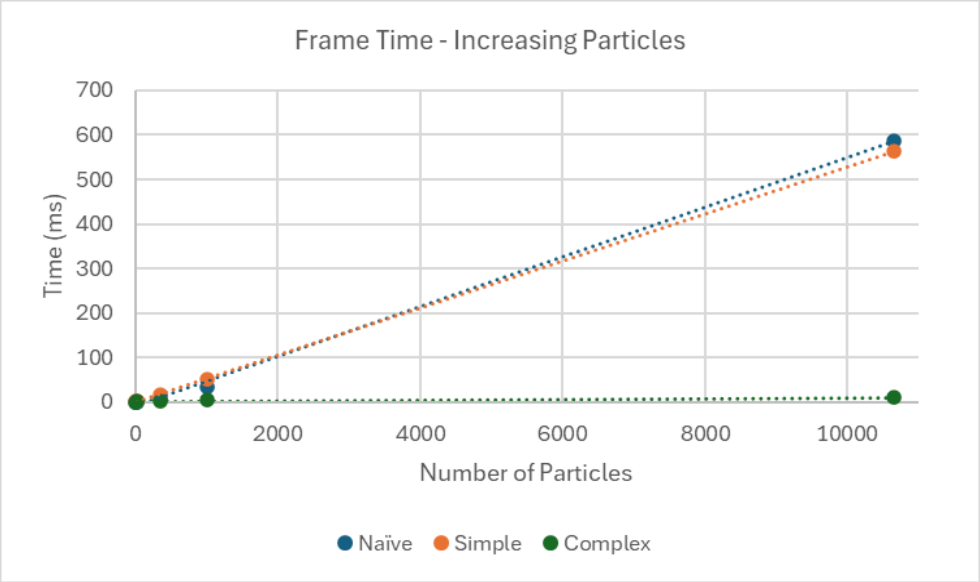

Figure 30 - Frame times in milliseconds as the number of particles is increased.

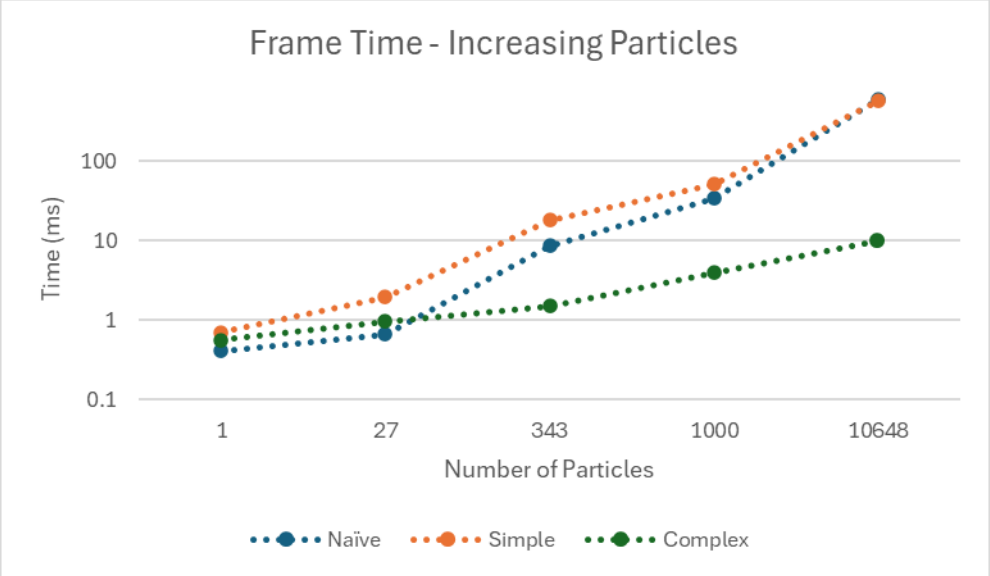

Figure 31 - Frame times in milliseconds as the number of particles is increased, logarithmic axis scaling.

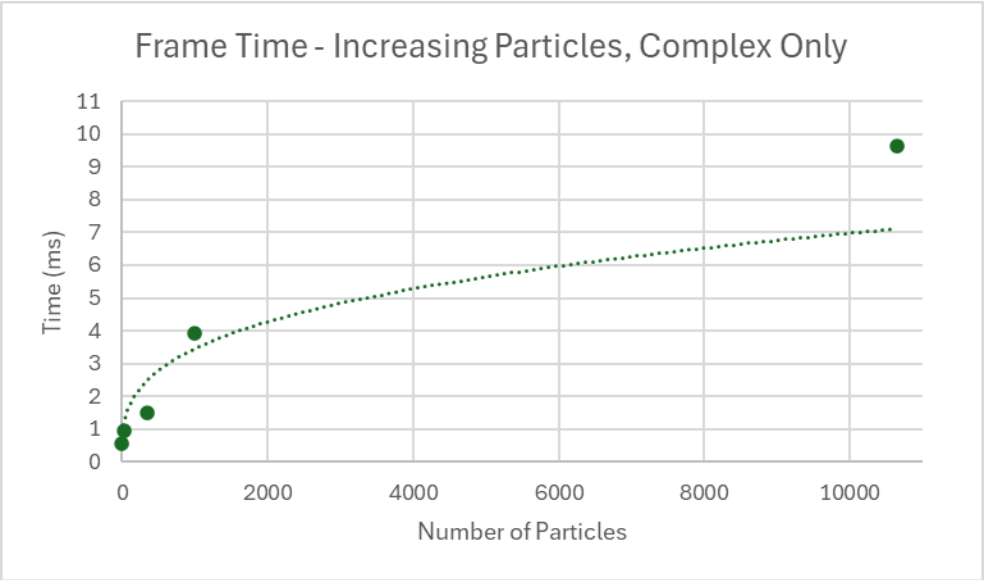

Figure 32 - Frame times in milliseconds as the number of particles is increased, complex pipeline only.

To put these results into perspective, the average frames per second inferred from the frame timings for each pipeline, when simulating 10648 particles, are shown in Figure 33.

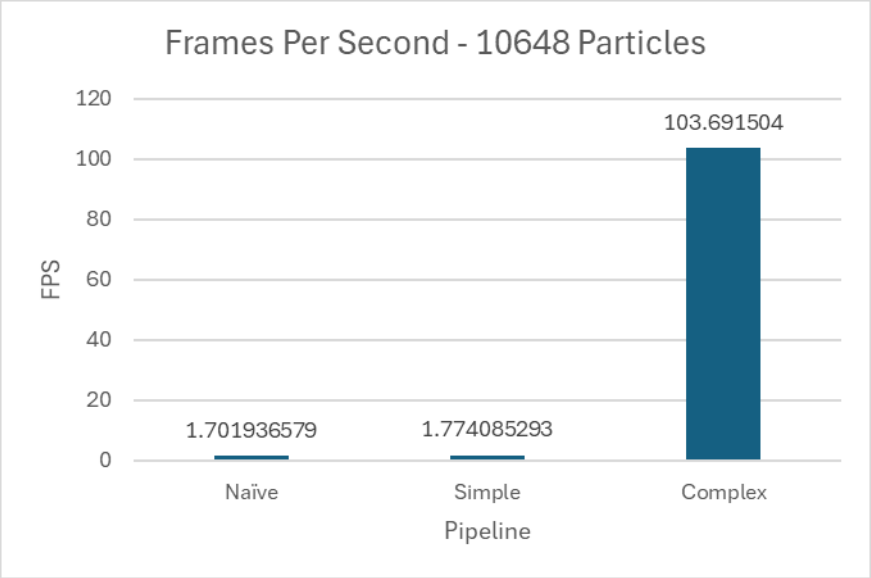

Figure 33 - Average frames per second for 10648 particles.

Breakdowns of the frame timings showing the percentage taken by individual workloads for the complex pipeline with 343 and 10648 particles are shown in Figure 34.

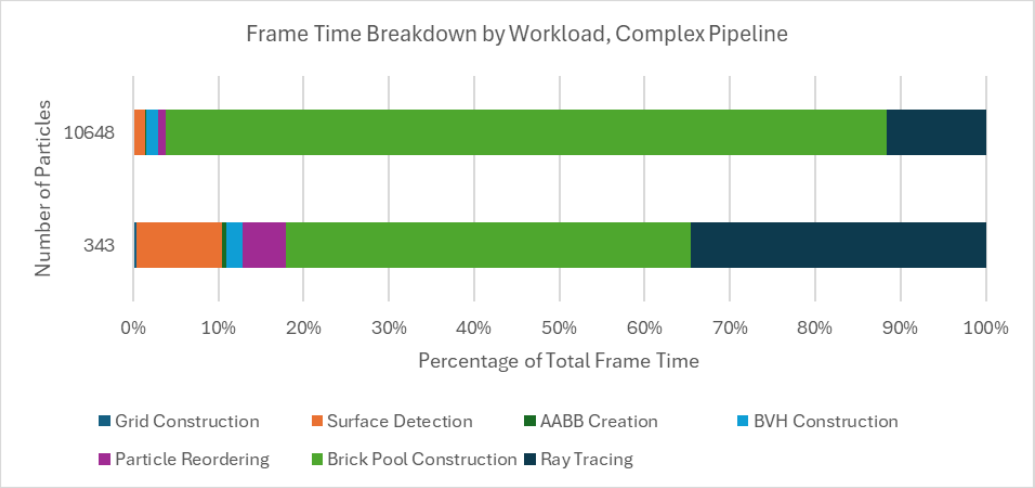

Figure 34 - Breakdowns of the total frame timings, showing the percentage taken by each GPU workload as particle number increases.

The total average memory overhead for each pipeline as the particle number increases is presented in Figure 35, and a breakdown of the memory overhead for the complex pipeline over a selection of particle numbers is shown in Figure 36. All measurements of memory are presented in megabytes, where 1 MB = 1000000 bytes.

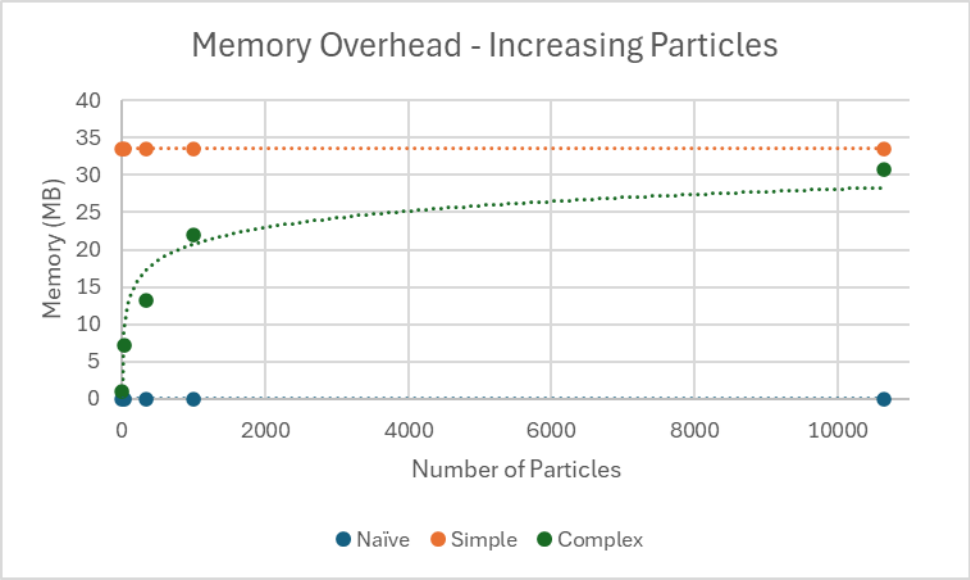

Figure 35 - Average memory overhead as particle number increases.

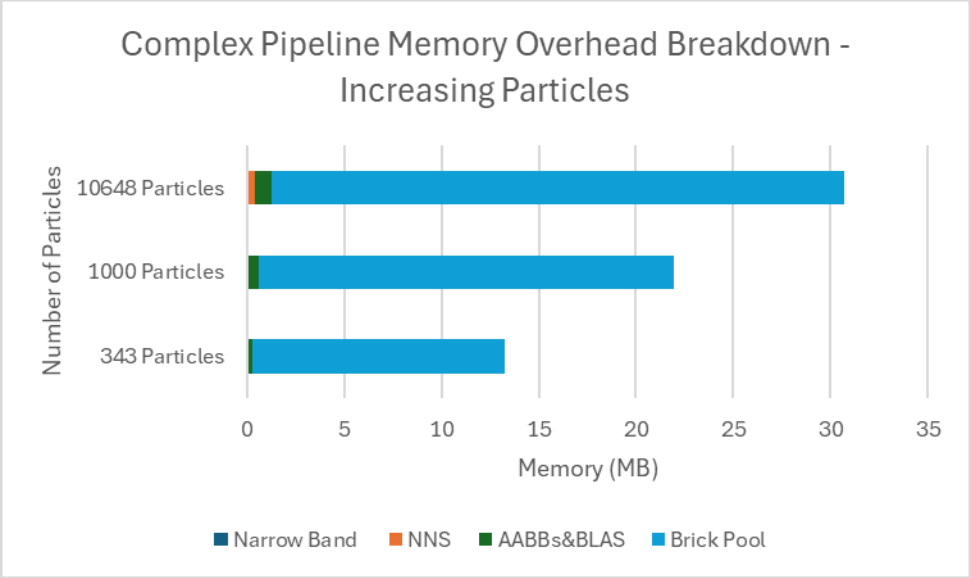

Figure 36 - Breakdown of the memory overhead for the complex pipeline over a selection of particle numbers.

4.2 Texture Resolution

The average frame times for the simple and complex pipeline as the specified texture resolution increases is presented in Figure 37, with the simple pipeline removed in Figure 38. The naïve pipeline does not make use of a 3D texture so is not included. Breakdowns of the frame timings showing the percentage taken by individual workloads for the complex pipeline with texture resolutions of 128^3 and 768^3 are shown in Figure 39.

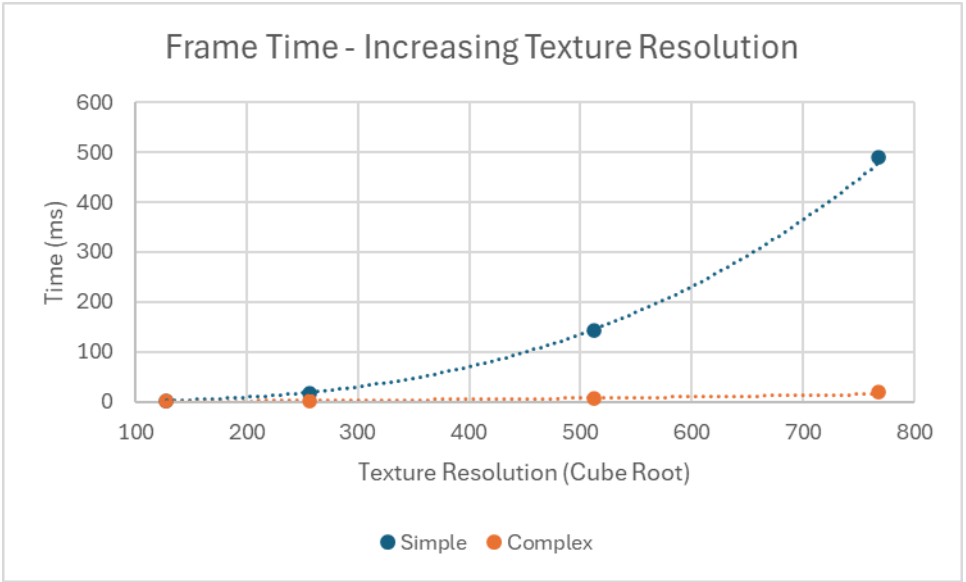

Figure 37 - Frame times in milliseconds as the specified texture resolution is increased.

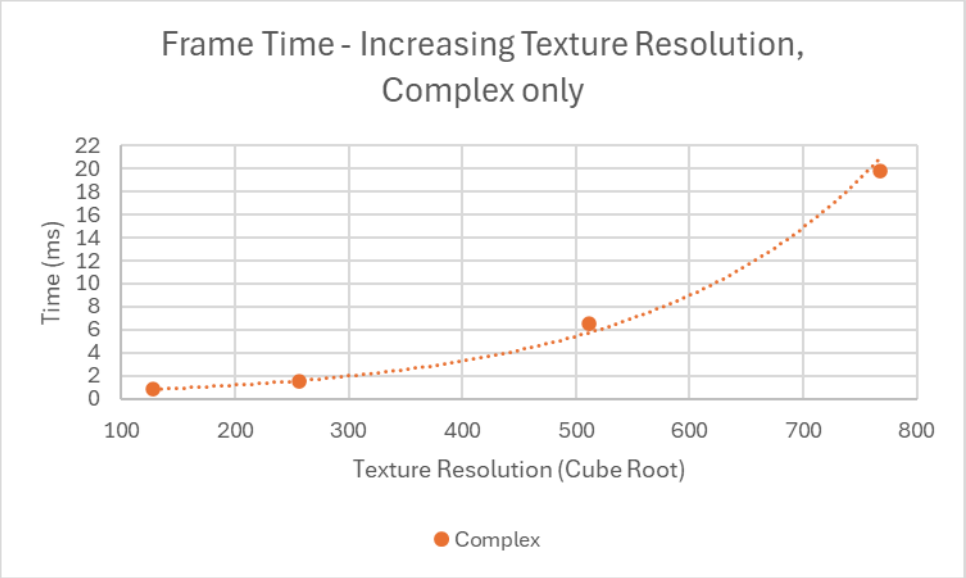

Figure 38 - Frame times in milliseconds as the specified texture resolution is increased, complex pipeline only.

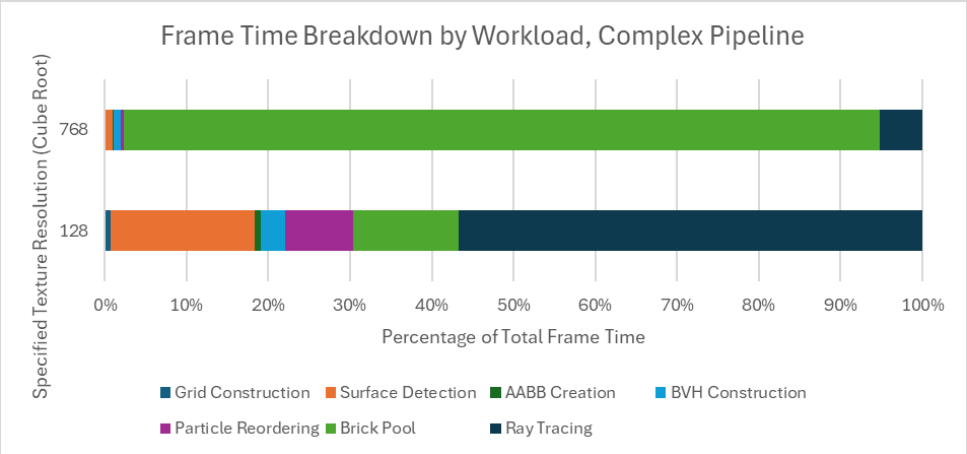

Figure 39 - Breakdowns of the total frame timings for complex pipeline, showing the percentage taken by each GPU workload as specified texture resolution increases.

The total average memory overhead for the simple and complex pipelines as the texture resolution increases is presented in Figure 40Figure 35, and a breakdown of the memory overhead for the complex pipeline for target resolutions of 512^3 and 768^3 is shown in Figure 41.

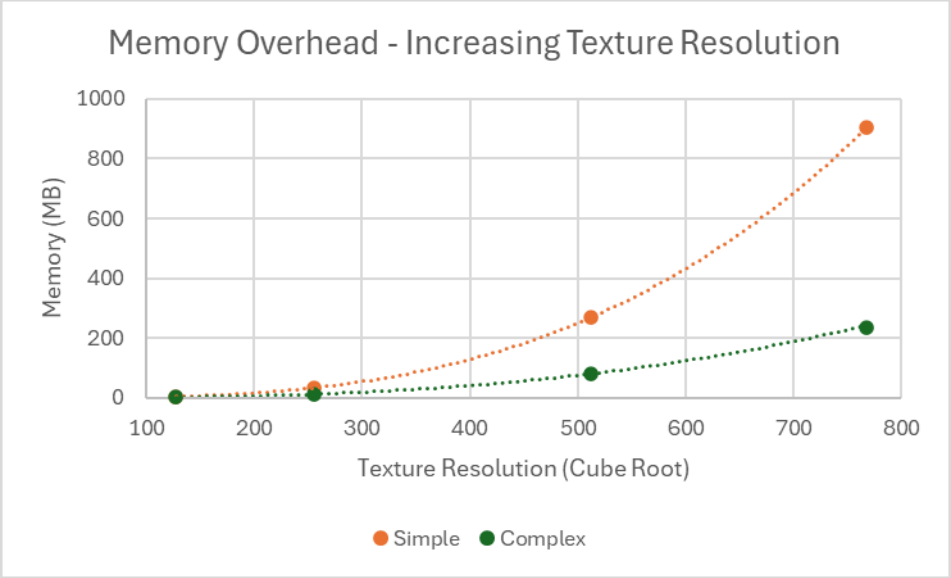

Figure 40 - Average memory overhead as target texture resolution increases.

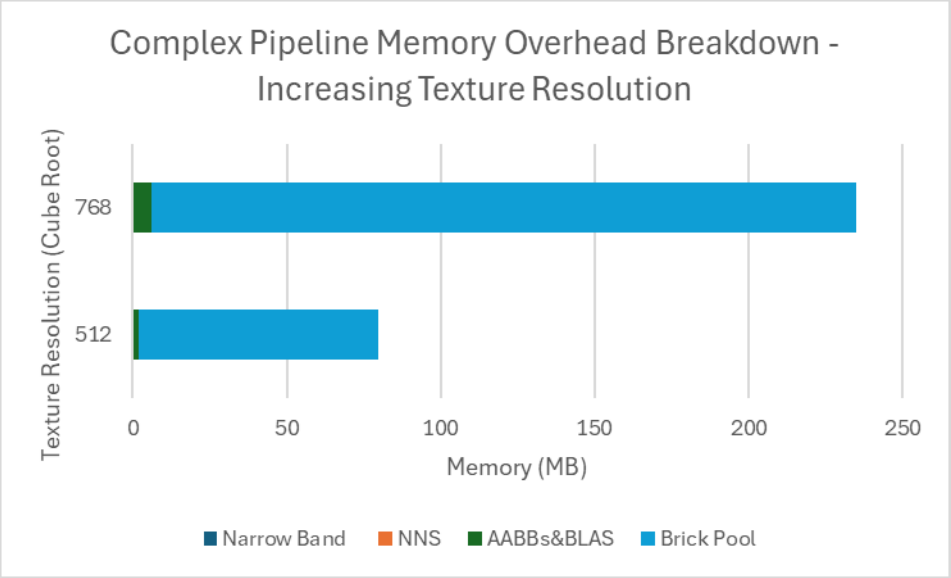

Figure 41 - Breakdown of the memory overhead for the complex pipeline for target resolutions of 512 and 768.

4.3 Screen Resolution

Figure 42 shows the average frame times for each pipeline as the number of pixels increases. Figure 43 shows only the timings for the complex pipeline. Memory overhead is not presented as view distance does not affect this.

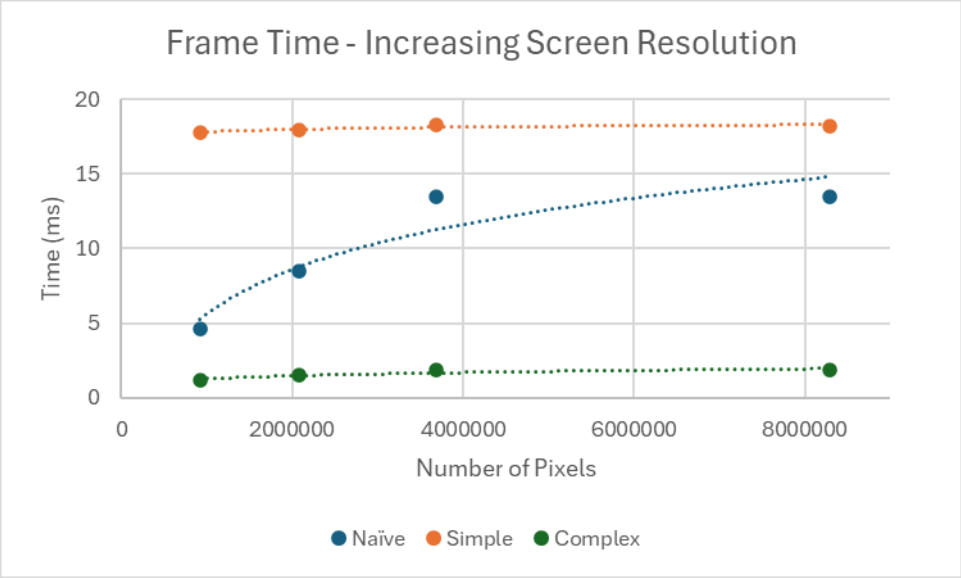

Figure 42 - Frame times in milliseconds as the number of pixels is increased.

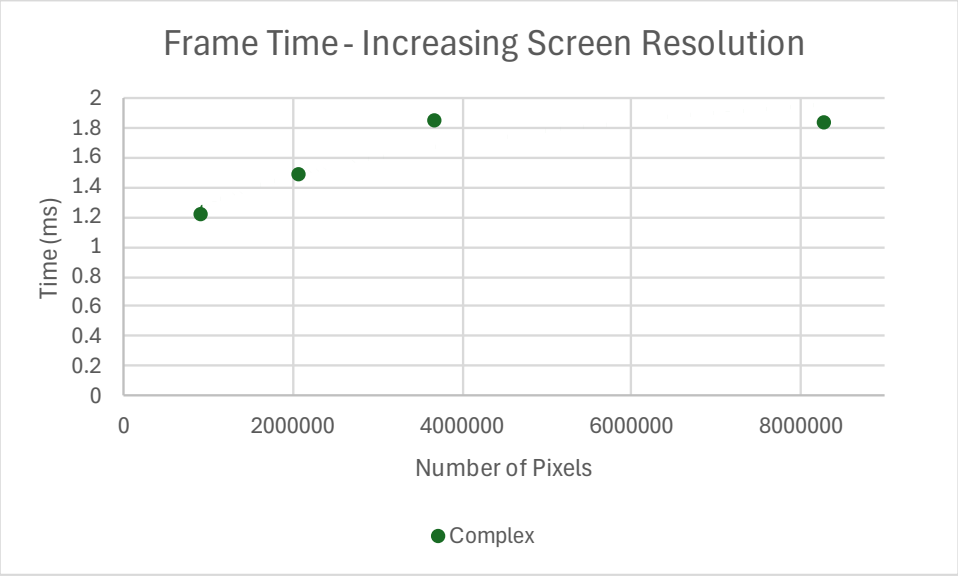

Figure 43 - Frame times in milliseconds as the number of pixels is increased, complex pipeline only.

4.4 View Distance

Figure 44 shows the average frame times for each pipeline as the distance of the viewpoint from the scene increases. Figure 45 shows only the timings for the complex pipeline. Breakdowns of the frame timings showing individual workloads for the complex pipeline at the lowest and highest view distances are shown in Figure 46. The same for the simple pipeline is shown in Figure 47. Memory overhead is not presented as it is unaffected.

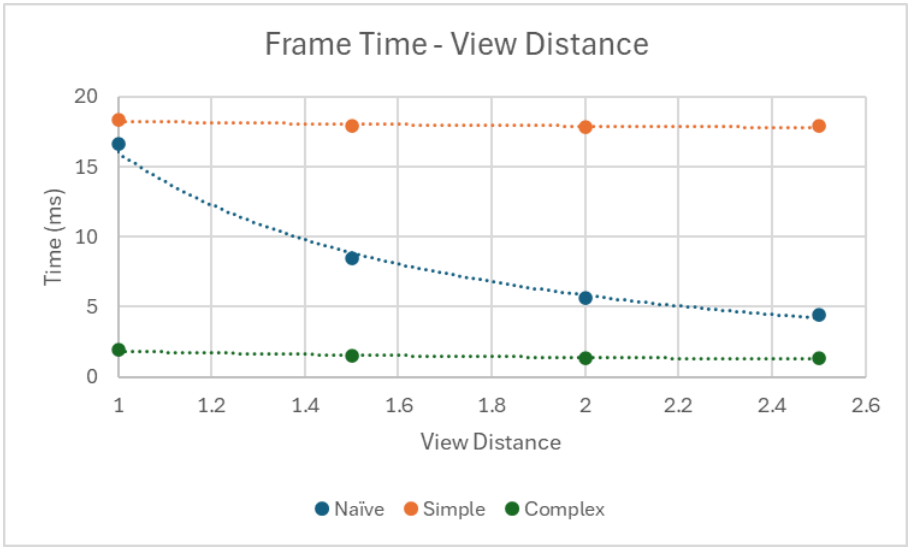

Figure 44 - Frame times in milliseconds as the view distance is increased.

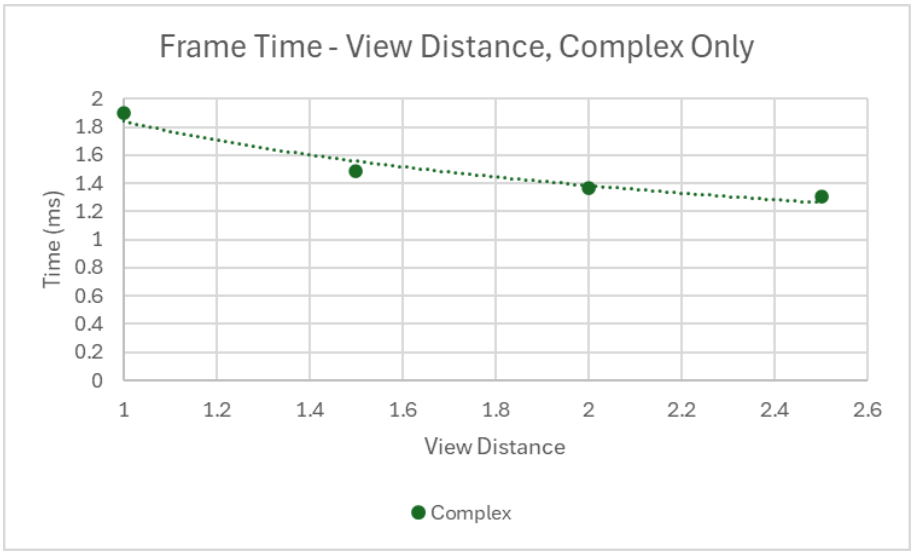

Figure 45 - Frame times in milliseconds as the view distance is increased, complex pipeline only.

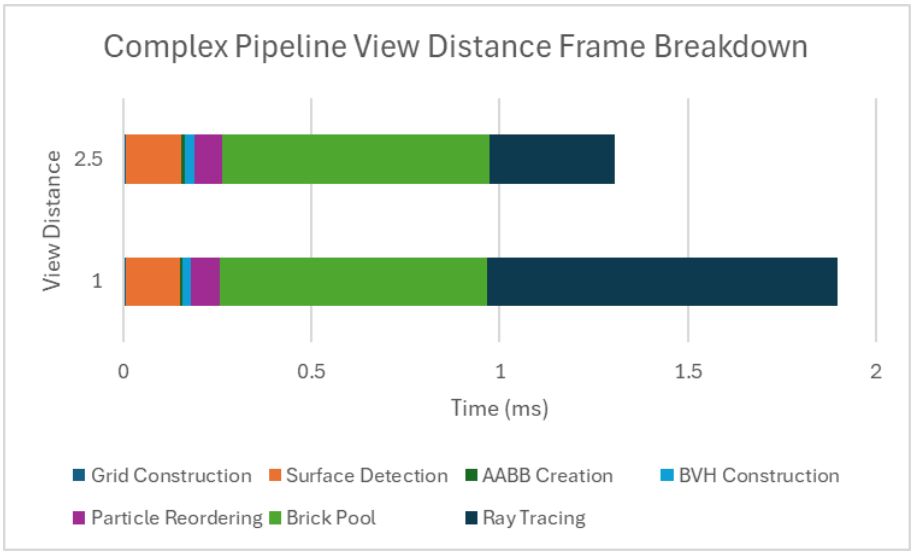

Figure 46 - Breakdowns of the total frame timings for complex pipeline, showing GPU workloads as view distance increases.

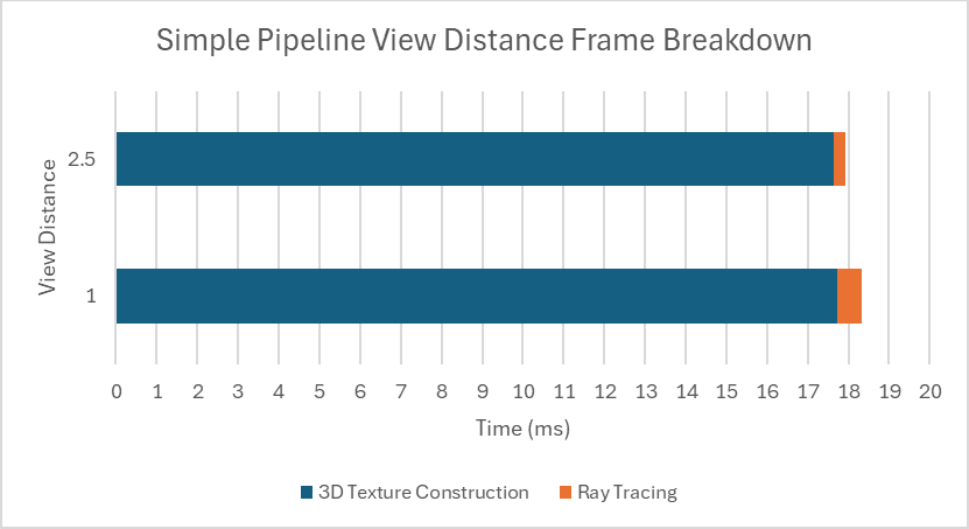

Figure 47 - Breakdowns of the total frame timings for simple pipeline, showing GPU workloads as view distance increases.

4.5 Scenes

Average frame times for all pipelines across the different scenes as seen in Figure 26, are shown in Figure 48. Breakdowns of the frame timings showing individual workloads for the complex pipeline across the scenes are presented in Figure 49.

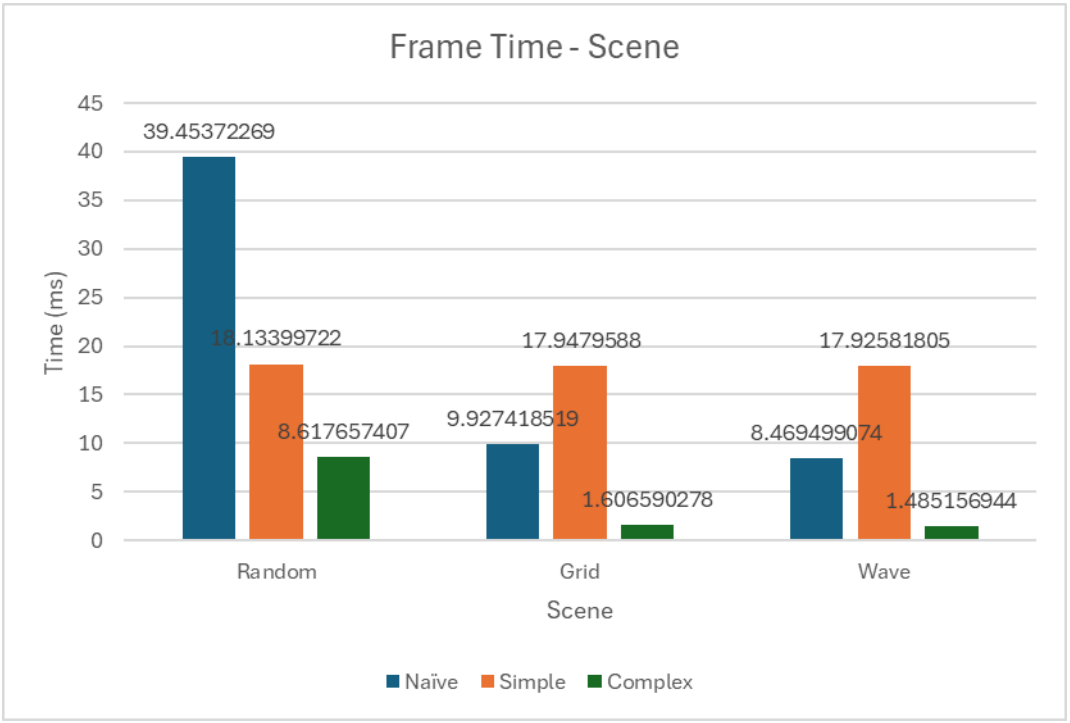

Figure 48 - Frame times in milliseconds across scenes.

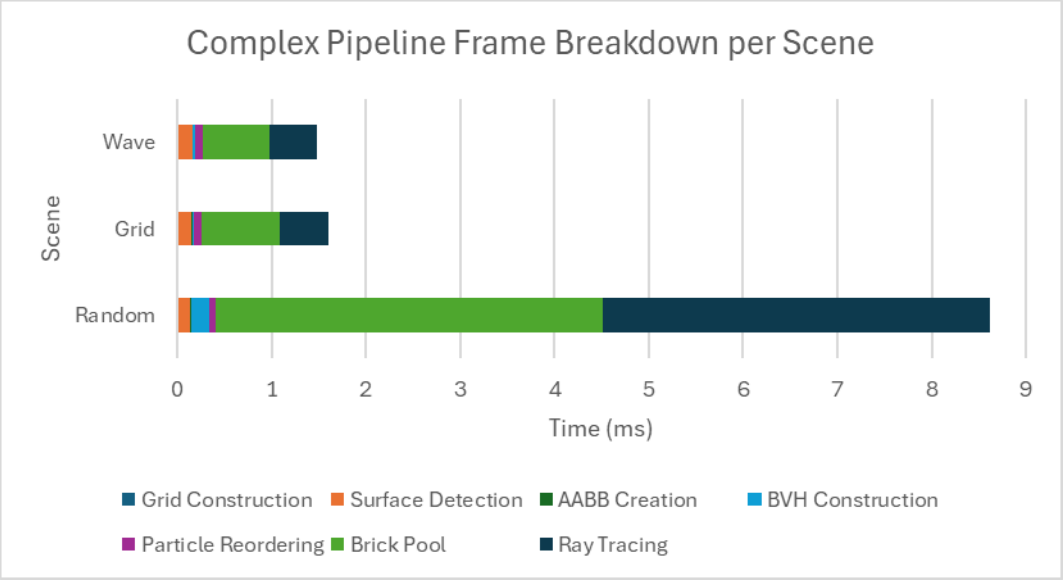

Figure 49 - Breakdowns of the total frame timings for complex pipeline, showing GPU workloads across scenes.

The total average memory overhead for each pipeline across the scenes are shown in Figure 50.

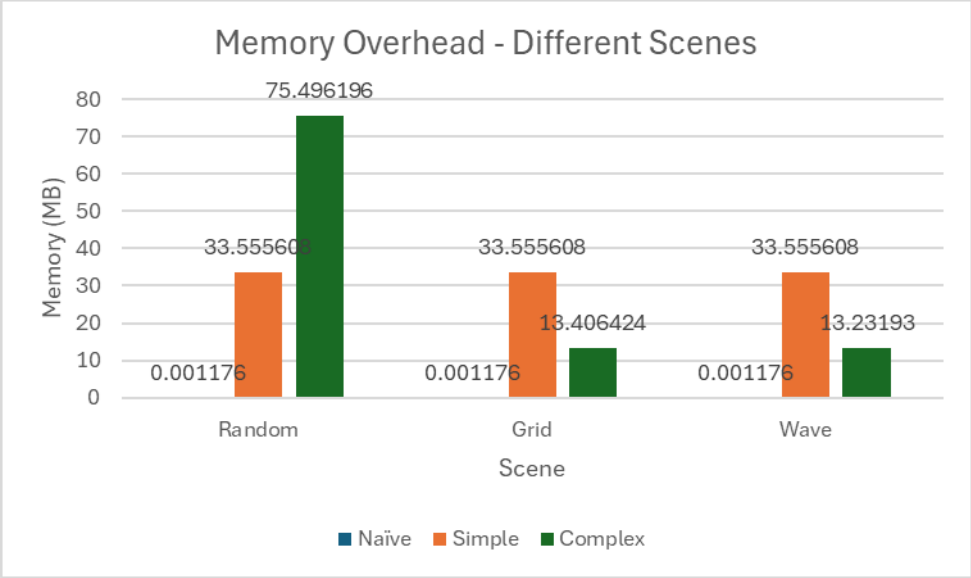

Figure 50 - Total average memory overhead for each pipeline across scenes.

4.6 Hardware

To gain a sense of how performance differs across different levels of hardware, tests using the default variable configuration were performed on the test systems listed in Table 1. The average frame times for all pipelines across the differing hardware are shown in Figure 51.

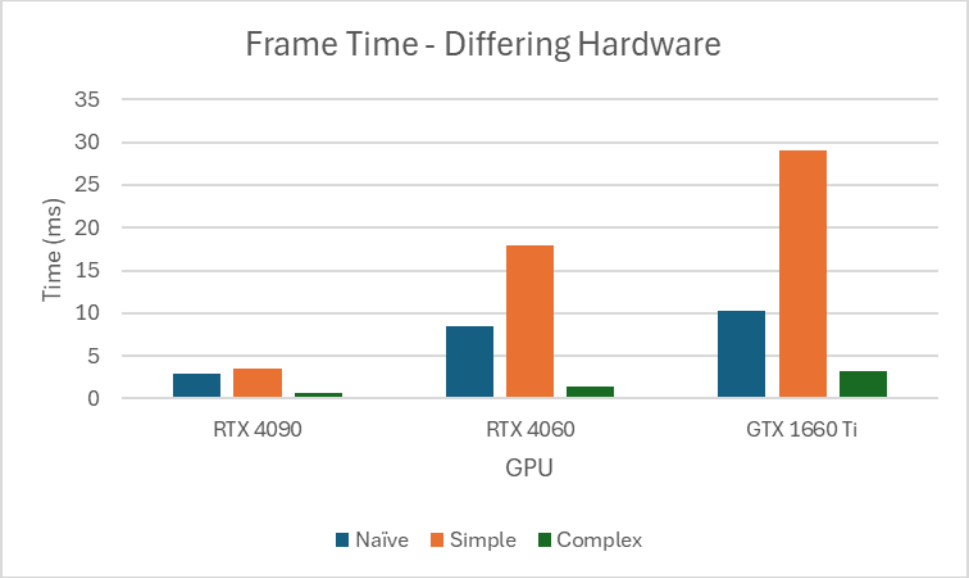

Figure 51 - Frame times for all pipelines across differing hardware.

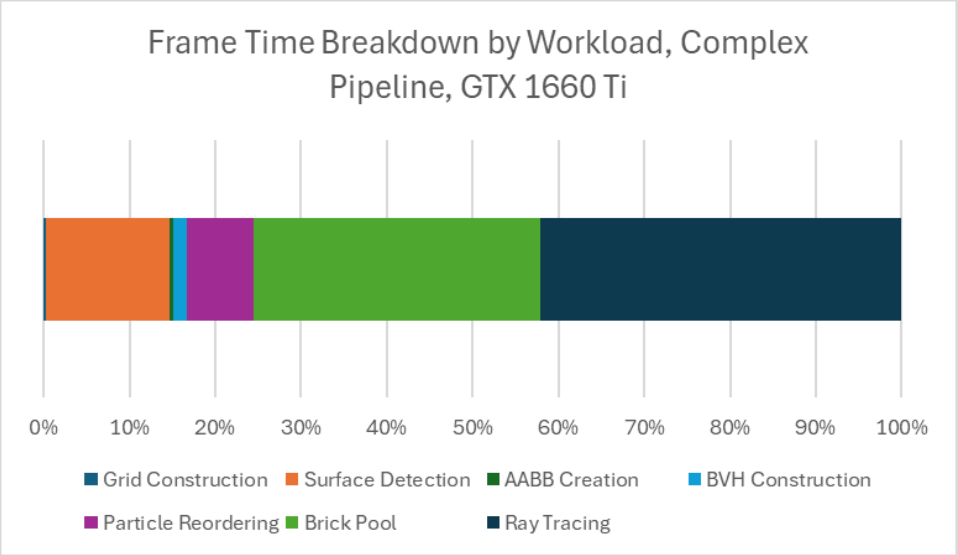

Figure 52 - Breakdowns of the total frame timings for complex pipeline, default configuration, running on a GTX 1660 Ti.

4.7 Visual Fidelity

The test scene as seen in Figure 27 was rendered at a screen resolution of 2560×1440 utilising the naïve pipeline. As it sphere traces the SDF analytically without discretisation, it follows that the naïve pipeline results in the highest fidelity rendering. The only variable likely to influence the visual fidelity is the granularity of the discretisation, i.e. the specified texture resolution. Renders were captured for both the simple and complex pipelines at all target texture resolutions and the differences between them and the naïve render were analysed. Figure 53 shows the differences observed, massively artificially enhanced for easier viewing,

and Figure 54 shows close ups of the same area in each render from the complex pipeline.

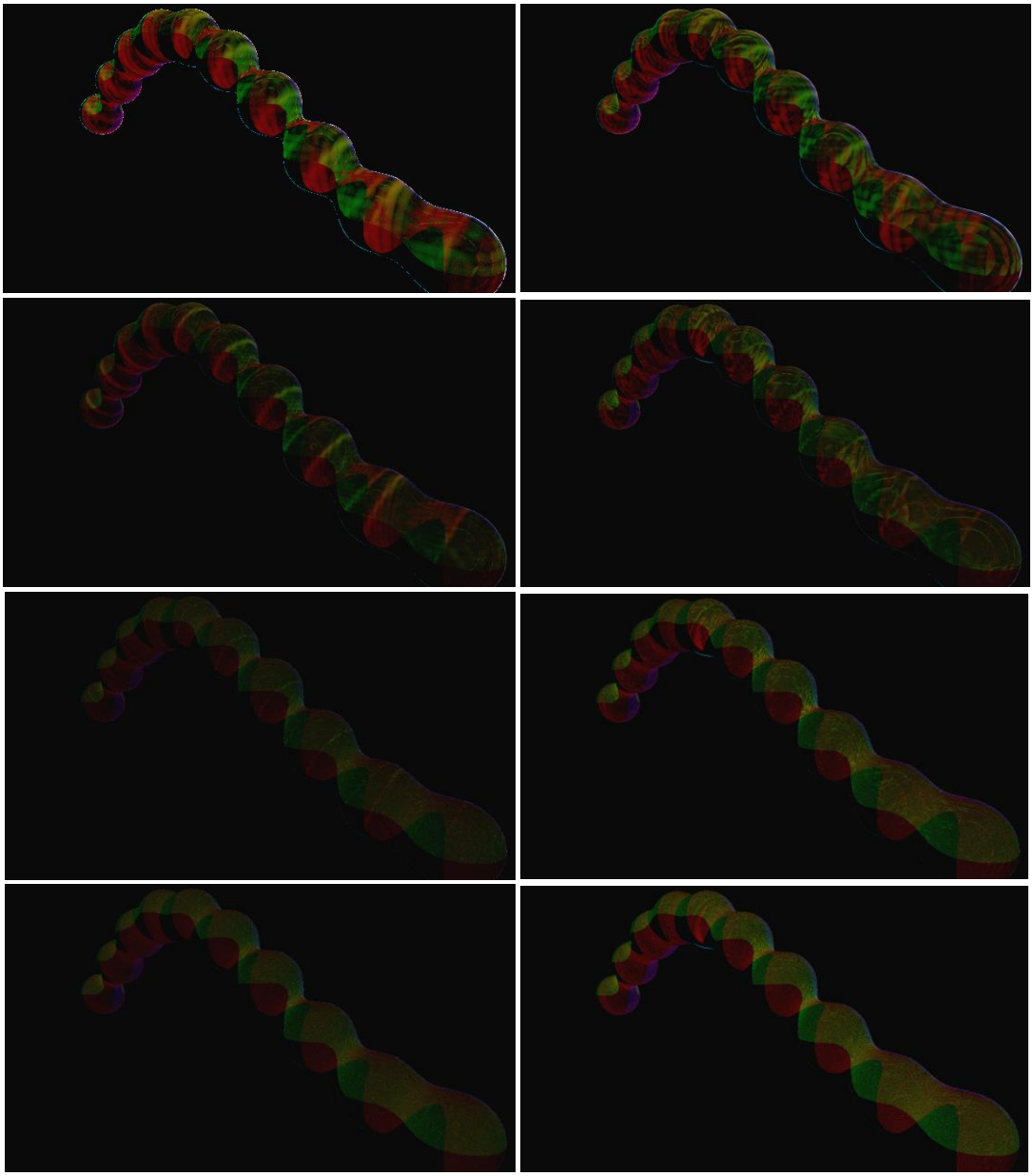

Figure 53 - Differences to the naïve render. Simple pipeline (Left) and Complex pipeline (Right). Texture resolutions from Top to Bottom: 128^3, 256^3, 512^3, 768^3.

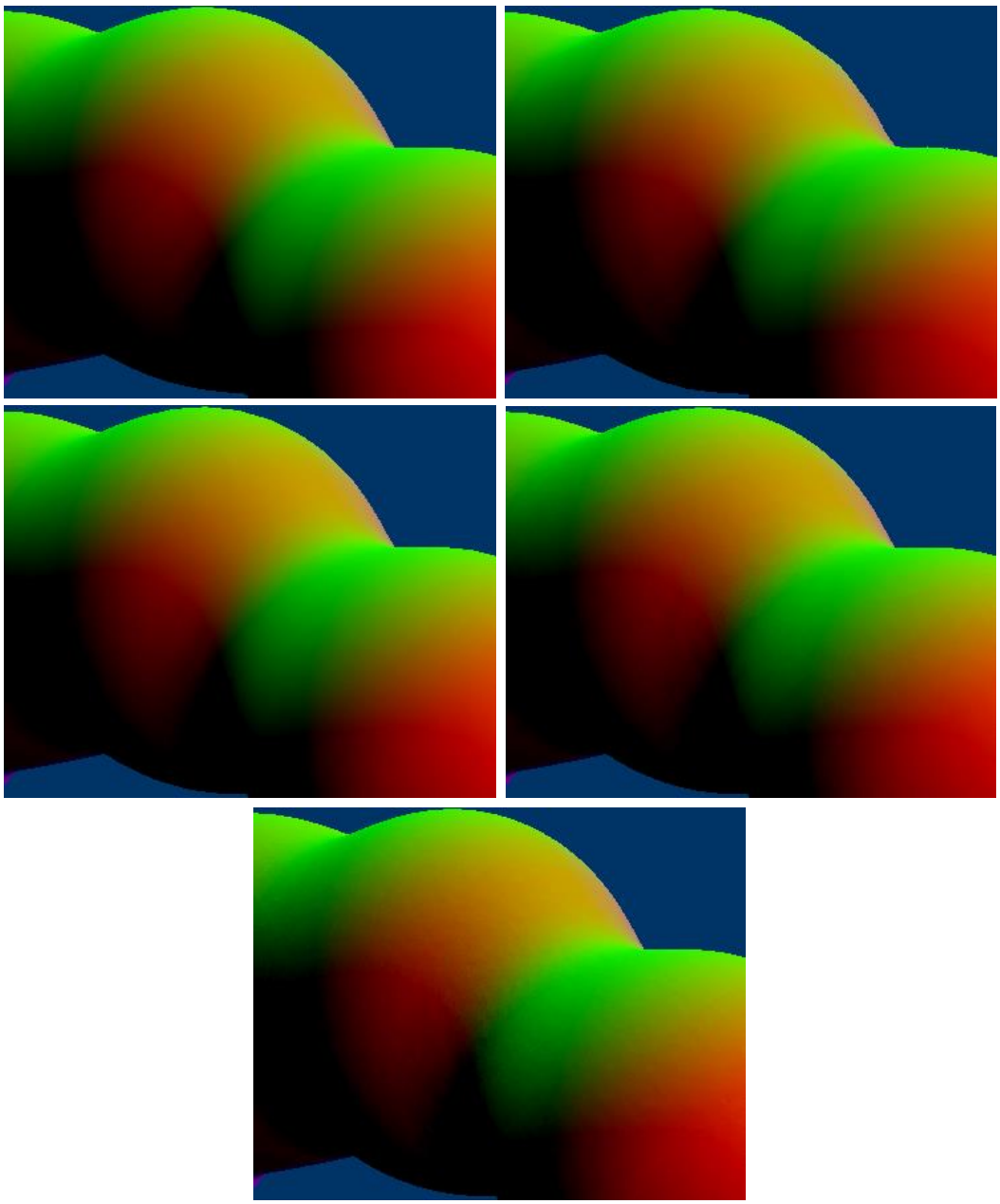

Figure 54 - Close-up of an area of the test scene. Texture resolutions: naïve (Top Left), 128^3 (Top Right), 256^3 (Middle Left), 512^3 (Middle Right) and 768^3 (Bottom).

Chapter 5 Discussion

The aim of this project was to evaluate the effectiveness of a pipeline combining a novel narrow band technique, an optimised nearest neighbour search and the construction of a sparse brick set data structure, based on the work of Nishidate and Fujishiro (2024), Liu *et al.* (2023) and Söderlund, Evans and Akenine-Möller (2022), compared to naïve and less advanced approaches. The factors being evaluated are whether the pipeline achieves the goal of reducing rendering latency and memory overhead effectively, without producing unsatisfying visual results. The study also aims to investigate the feasibility of the pipeline for use in modern games. It was hypothesized at the end of Section 2.5 that the produced pipeline would indeed prove an effective means of improving performance and reducing memory overhead.

The scope of testing was limited to investigating the effects of varying the variables listed in Table 3. While varying the configuration of the spatial grid used, for example the number and sizes of the blocks and cells, would no doubt affect the performance and memory overheads involved, any improvement observed while utilising any configuration is enough to demonstrate the pipelines effectiveness over the less advanced techniques. Furthermore, as explained in Section 3.3.1, the radius of influence considered by the smoothing function which blends the signed distances to particles is limited by the size of cells, thus changing the grid configuration will affect what is possible in terms of the produced geometry. It is therefore suggested that developers utilising the pipeline first decide upon a desired radius of influence h and the world space bounds, which will likely depend on the art direction and the underlying fluid simulation, and determine the grid configuration accordingly such that the cell side length = h.

5.1 Particles and Texture Resolution

Observing how the complex pipeline fares compared to the simple and naïve pipelines when the number of particles is increased, as seen in Figure 30, it is immediately clear that the complex pipeline offers significant improvement to the rendering latency. The frame times of both the naïve and simple pipelines scale linearly with the particle number, whereas Figure 32 shows logarithmic scaling in the case of the complex pipeline, staying below 10 milliseconds for all values

tested. This is consistent with the findings of Nishidate and Fujishiro (2024) and it is expected that this trend will continue as even higher numbers of particles are simulated; an encouraging result in terms of applicability to use in games. The values displayed in Figure 33 show that in the best case, the framerate achieved by the complex pipeline inferred from the frame timings (by converting to seconds and raising to the power of -1) demonstrate a speedup of 5992.56% over the naïve method when rendering a simulation of 10648 particles.

Naturally, this does not account for all other processes which must occur over the course of a frame (such as reading back values from the GPU, copying the result of ray tracing to the back-buffer, rendering the user interface, presenting the frame etc), and at almost 10 milliseconds, over half of the 16.67 millisecond budget for achieving 60 frames per second is consumed. As frame times will only increase as the particle number increases, it is therefore likely that this pipeline is best suited to small-scale fluid simulations, for example water in a glass, rather than large-scale effects such as an open ocean. Nishidate and Fujishiro (2024) were able to render simulations of hundreds of thousands of particles in under 6 milliseconds using a marching cubes implementation, suggesting that while the complex pipeline implemented for this project offers significant gains over the less advanced methods of non-polygonal rendering, polygonal rendering using an intermediary mesh is still the most performant option available to developers.

It can be observed from Figure 34 that in most cases the construction of the brick pool is the largest bottleneck, with this process's share of the frame time increasing as particle number increases. It was found that similarly, the 3D texture generation is the bottleneck for the simple pipeline in all cases, as seen in Figure 47. However, unlike in the simple pipeline, the time to construct the brick pool is seen to be curtailed by the narrow band and NNS techniques minimising the number of SDF values evaluated, and the number of particles considered during evaluation. This observation is backed up by the peak SM throughputs displayed in Figure 29 which suggest less strain on the processors while using the complex pipeline. The improved cache coherency arising from the efficient reordering of particle data for use by the NNS algorithm further improves SDF evaluation times by increasing the cache hit rate as seen in Figure 28, confirming the work of Liu *et al.* (2023).

As made clear by Figure 31, the simple pipeline is the slowest of the three under the conditions tested, only overtaking the naïve method in the most extreme case. This is likely due to the number of voxels being written to the 3D texture; for a texture of resolution 256^3, there are 16777216 such voxels, and indeed, further examination of the data shows the simple pipeline to consistently exhibit the highest LSU throughputs during SDF evaluation. The simple pipeline could perhaps be optimised then by utilising groupshared memory to write to the texture in chunks, reducing global memory accesses. It is speculated that the complex pipeline would remain the fastest if this is the only improvement made, however, it would be of interest to measure the result of applying the narrow band and NNS techniques for SDF evaluation, while retaining the dense 3D texture. It was speculated that the use of large threadgroup sizes for the simple pipelines 3D texture generation would be another limiting factor due to contention for SM resources, however no stalls due to GSM allocation were observed and those due to warp allocation are very low; experimenting with thread group sizes both in this case and in the complex pipeline, for example during the grid construction stage, is still a potentially worthwhile avenue of research. It should also be noted that the effectiveness of the simple pipeline over the naive pipeline is largely dependent on view distance and screen resolution, which will be discussed further, but is unaccounted for in the figures discussed above.

Similarly to the frame timings, the memory overhead present while utilising the complex pipeline is seen to increase logarithmically as the number of particles increases, with the brick pool consuming the most memory by far as seen in Figure 36. Since voxels representing SDF values are only stored in bricks close to the fluid surface, the increase in memory usage is less to do with the increased particle number and more due to the increasing fluid surface area as the simulation takes up more of the world space. As will be further discussed, the memory overhead can exceed that of the simple pipeline in the worst cases, however this does not occur in the simulated wave scene, which is the closest in behaviour to a typical SPH simulation as would be used in a video game. The naïve pipeline has a very low memory overhead; however, the trade-off made when using the complex pipeline is likely worth it given the impressive performance gains.

As can be observed from Figure 37, Figure 38, Figure 40 and Figure 41, both the complex and simple pipelines demonstrate increasing frame times and memory overhead as the specified texture resolution increases, driven by the need to fill more voxels with SDF values, which is confirmed to be the limiting factor by Figure 39. The graphs appear to show exponential growth; however, the horizontal axis shows the cube root of the texture resolution (i.e. for a resolution of 256^3, the value is 256), thus the growth is in fact linear in relation to the overall texture size. As explained in section 3.3.2, the brick pool does not actually match the specified texture size, but enough voxels are stored to match the granularity in world space, thus the brick pool's size is significantly lower than that of the simple pipeline's dense 3D volume yet still scales linearly as the target resolution increases.

5.2 View Distance and Screen Resolution

Observing Figure 44 and Figure 45, it can be seen that the frame times for both the complex and naïve pipelines gradually decrease logarithmically as the distance from the viewpoint to the fluid surface increases. This is due to the reduced latency of ray tracing, because as the scene takes up less screen space, less rays intersect with the scene and thus there is less work to be done; Figure 46 confirms this, as it can be seen that ray tracing is the only reduced workload as the distance increases.

The same is true of the simple pipeline, as seen in Figure 47, however the frame times are clearly bottlenecked by 3D texture construction and the reduced ray tracing workload has little impact, thus the frame times stay roughly constant. Consequently, since frame times for the naïve pipeline increase as the scene fills more screen space, the simple pipeline does eventually out-perform the naïve implementation when view distance becomes small enough. Looking at the data from which Figure 46 and Figure 47 were produced, it is found that the time for ray tracing is in fact lower for the simple pipeline than the complex pipeline (at a view distance of 1, the time for ray tracing in the simple pipeline was 0.60ms compared to the complex pipeline's time of 0.93ms), which backs up the findings of Söderlund, Evans and Akenine-Möller (2022) that sphere tracing through a dense volume tends to be faster than BVH traversal for simple scenes, however it is clear that the impact of BVH traversal is small while the large reduction in memory overhead makes the SBS a worthwhile compromise; especially for use in

games where memory usage is a critical issue. If memory usage is not a concern however, it could be the case that implementing the technique earlier speculated on, applying the narrow band and NNS methods to the dense 3D volume generation, could produce even lower frame times than the current complex pipeline.

It could also be investigated how use of a lookup table for accessing bricks during ray tracing affects the performance of the complex pipeline; currently, bricks are accessed via on-the-fly index calculations. In most tests the ALU throughput was around 40-45%, whereas the LSU throughput was around 12%. Therefore, it may be the case that use of a lookup table would balance the throughputs, putting less strain on the ALU, potentially decreasing rendering latency at the cost of additional memory overhead. Another possible performance increase could be realised by employing a method inspired by Crassin *et al.* (2009) and Laine and Karras (2010), where a brick would be evaluated only when a ray intersects with its AABB. This would decrease the workload overall, however when bricks are required and not yet evaluated, the brick pool would need to be re-constructed for a second time during the frame, which could present itself as sudden drops in the framerate and lead to a negative player experience.

It could be anticipated that performance would degrade linearly when screen resolution increases as more rays intersect the scene, however, curiously the frame times instead increase logarithmically for the naïve and complex pipelines as seen in Figure 42 and Figure 43. Initially the increase in rays leads to an increase in active processing units (using the complex pipeline, occupancy increases from 43% to 50% during ray tracing when screen resolution increases from 720p to 1440p), but this levels out in line with the timings (occupancy remains at 50% at a resolution of 4k), suggesting warps are being efficiently scheduled, spreading work evenly among the SM units, which could explain the lack of linear time increase. This would be a benefit for modern games, whose users likely have a wide range of screen resolutions.

Similarly to when view distance is increased, the frame times for the simple pipeline remain roughly constant as the cost of ray tracing is not a bottleneck (Figure 47).

5.3 Scenes

The different scenes tested demonstrate how different particle behaviours affect performance and memory overhead. As seen in Figure 48 the simple pipeline performs roughly equally across all scenes, as performance is bottlenecked by the 3D texture resolution generation which fills voxels for all points in space regardless of the fluid behaviour. The performance of the naïve and complex pipelines seems to be dependent on the fluid surface area and the intricacy of the geometry. The *Grid* scene represents the simplest shape a collection of particles can form and thus can be viewed as a baseline for analysis. The *Wave* scene compresses the particles to imitate the forces of gravity and surface tension, reducing surface area; accordingly, surface area is reduced, and performance is marginally better as less rays come close to intersecting the surface. In the case of the *Random* scene, there is much more surface area as particles are more spread out with many small gaps between the geometry. This increase in the complexity of the geometry gives rise to an increase in ray tracing time as more sphere tracing steps are required to safely traverse the scene. In the case of the complex pipeline, frame times are also increased as many more bricks are required, as the simulation fills most of the world space, which is backed up by Figure 49.

The *Random* scene also demonstrates the case where memory overhead for the complex pipeline can exceed that of the simple pipeline, as seen in Figure 50. This is because voxels are required for almost the entire world space, negating the advantages of the sparse brick set. The additional voxels required for adjacency data, as explained in Section 3.3.2, are why the brick pool exceeds the size of the simple dense 3D texture in this case. The additional memory overhead is likely worth it, however, considering the performance gains. One potential way of removing the overhead introduced by adjacency voxels in the brick pool is instead of relying on hardware linear interpolation to produce smoothed values during sampling, the relevant voxels could be point sampled and manually interpolated in during sphere tracing. This would likely harm performance however as hardware interpolation is faster. An alternative to manual smoothing during each sphere tracing step would be to employ a scheme similar to that of Quispe and Paiva (2022), applying smoothing kernels to the contents of each brick as a post-processing step after SDF evaluation. This may well produce artifacts due to inconsistencies at brick boundaries, however.

5.4 Hardware

The GPUs listed in Table 1 constitute a range of widely available consumer hardware. The GTX 1660 Ti represents a low-end, budget-friendly choice for casual gamers. The RTX 4060 is a modern, popular mid-range option and the RTX 4090 represents the top-tier, highest performing consumer GPUs. From Figure 51 it is observed that performance scales accordingly.

The RTX 4090's increased compute power seems to remove much of the issues bottlenecking the simple pipeline, with the complex pipeline only performing marginally better. Conversely, the GTX 1660 Ti has limited processing units, which affects its ability to quickly evaluate and store SDF values in both the simple and complex pipelines, almost reaching full occupancy during these stages. In addition to the slower brick pool construction, the complex pipeline is bottlenecked by the cost of ray tracing, as seen in Figure 52, as the GTX 1660 Ti does not have dedicated ray tracing hardware and must instead perform software BVH traversal through driver support. It was found that ray tracing on the GTX 1660 Ti was the only recorded instance of warp stalls due to GSM allocation, likely related to the driver implementation of BVH traversal.

These findings suggest that the complex pipeline offers diminishing returns on performance with better hardware, however the reduced memory overhead and improved performance on lower-end equipment likely makes its use attractive for games reaching a wide audience.

5.5 Visual Fidelity

As explained in Section 3.4.2, visual fidelity will be defined as how closely the fluid geometry and surface normals resemble those produced by the naïve implementation. Figure 53 presents the differences in the images produced by both the simple and complex pipelines from the naïve render, enhanced for ease of viewing. It appears that the simple pipeline produces both the most and least pronounced differences, at texture resolutions of 128^3 and 512^3 respectively. The results from the complex pipeline fall somewhere between the two extremes, presenting a satisfactory middle ground. There is clearly noise present in the normals, however, especially visible for the higher texture resolutions, which is not as pronounced in the renders from the simple pipeline. Figure 54 shows a comparison of how the normals appear without enhancement, and evidently the

noise is only immediately apparent in the renders using 512^3 and 768^3 texture resolutions. The noise is much less visible for the 128^3 and 256^3 texture resolution renders, however could present itself in specular highlights depending on shading methods. With constantly moving, transparent water this is not likely to be noticed by the majority of players.

The thicker edges around the geometry using the lowest resolution texture point to an inaccuracy in the shape of the geometry caused by the lower granularity; each voxel represents a larger portion of the world space, thus the sphere tracing is less precise and ray-isosurface intersections are found too early. This is almost negligible when the target texture resolution is 256^3, and this combined with the less distracting normals and performance results discussed earlier point to 256^3 being an ideal target resolution, producing satisfactory visual fidelity.

The excess noise observed in the normals is likely related to the limited step size explained at the end of Section 3.3.3. One way of mitigating this would be to increase the layer of adjacency data within each brick in the brick pool to 2 voxels thick, allowing proper sampling without crossing brick boundaries. This would increase the memory overhead considerably, however, and given that the visual fidelity is already satisfactory, is likely not worth the trade-off. The technique discussed earlier whereby adjacency data and hardware interpolation is removed in favour of applying a smoothing kernel to the bricks may potentially prove effective. Another possible solution would be a hybrid of the naïve and complex approaches; instead of discretising and storing SDF data in a 3D texture, only the brick AABBs and the resulting BVH would be created, and the SDF could be analytically sphere traced within a given AABB while still utilising the NNS algorithm. Further research is required to determine the merits of this method.

An additional point of consideration is that while interacting with the application in general, a flickering effect can sometimes be observed to affect the geometry when static particles are grouped closely together. This is likely due to the smooth minimum function used during SDF evaluation, as it produces slightly different results depending on the order in which particles are blended. As the particle data reordering step is non-deterministic in terms of the final particle order within each cell, the final SDF can change between frames. This is unnoticeable

while particles are in motion and a static simulation is unrealistic for fluids in games, so this is mainly a moot point.

Chapter 6 Conclusion and Future Work

This study aimed to design and implement a pipeline for the non-polygonal rendering of particle-based fluid simulations, combining a novel narrow band technique proposed by Nishidate and Fujishiro (2024), an optimised nearest neighbour search suggested by Liu *et al.* (2023) and a sparse brick set data structure presented by Söderlund, Evans and Akenine-Möller (2022). The implemented pipeline was to be evaluated in its effectiveness at reducing rendering latency and memory overhead when compared to less advanced methods.

As such, an application was developed incorporating all three of these techniques, along with two additional pipelines. The *naïve* pipeline directly sphere traces (Hart, 1996) the isosurface of a signed distance field (SDF) representing the fluid body analytically, re-evaluating the function many times each sphere tracing iteration. The *simple* pipeline attempts to optimise this by precomputing the SDF and storing it in a 3D texture; a dense volume of SDF values representing the whole world space. The *complex* pipeline utilises the abovementioned techniques.

Unit testing was performed to gather performance and memory overhead data for each pipeline under varying conditions. It was found that the complex pipeline offers an improvement over both the naïve and simple pipelines in almost all cases, with frame times scaling logarithmically as both the number of particles simulated and the resolution of the precomputed SDF increases, as opposed to linear scaling observed in the other pipelines. In the best case, the complex pipeline demonstrated a performance gain of nearly 6000%. Memory overhead likewise tends to scale logarithmically and remains lower than that of the simple pipeline in most cases. The complex pipeline's memory overhead does exceed that of the simple pipeline in cases where the fluid surface fills most of the world space bounds, however this is not a realistic scenario, and the performance gains are likely worth the compromise. Testing was conducted across a range of consumer GPUs and the complex pipeline was found to perform well at all tiers, with the most noticeable gains in performance for low-end hardware. The visual fidelity was investigated by comparing the surface normals and geometry produced by the complex pipeline against those of the naïve pipeline, and it was found that while there are slight artifacts and noise present, they are likely not so harsh as to disrupt player immersion.

These results have shown that the techniques incorporated into the implemented pipeline complement each other wonderfully and the pipeline is indeed effective at reducing rendering latency and memory consumption when compared to the naïve and simpler methods. The pipeline can feasibly be applied to use in modern video games for small-scale fluid effects, however comparison to the results of Nishidate and Fujishiro (2024) suggests that polygonal rendering through use of the marching cubes (MC) algorithm is likely still the most performant option for this.

There are several avenues of potential further research. Firstly, a more in-depth comparison of the implemented pipeline with polygonal, MC-based methods is needed to determine how attractive the pipeline is as an option overall.

While the research has demonstrated an effective prototype, the current implementation is probably not as well optimised as it could be. Future work could investigate determining optimal grid configurations for different simulation conditions. Experimenting with different threadgroup sizes throughout the various shader stages may also uncover performance gains. During the grid and AABB construction stages, surface block and cell counts are read back from the GPU to the CPU; the effect of dispatching shaders indirectly without waiting for data readback could be investigated. Improved use of groupshared memory could potentially expedite certain stages such as the surface cell detection stage which currently performs many calculations per thread to locate neighbouring cells.

Future research could consider potential ways of improving ray tracing performance. A lookup buffer for accessing bricks is a possible option, at the cost of additional memory overhead. Another approach may be a view-dependent scheme similar to that of Crassin, *et al*. (2009) and Laine and Karras (2010), constructing bricks only when required. Another avenue of research could be to investigate alternatives to sphere tracing for locating ray-isosurface intersections.

Memory overhead could perhaps be further reduced by removing adjacency data stored in each brick and instead performing manual software interpolation during sampling, or by applying blurring kernels to the volume, similar to the work of Quispe and Paiva (2022). Currently the implementation stores SDF values as 16-bit signed normalized values, however intelligent formatting of distance values could instead see use of 8-bit values.

Finally, there are some possible hybrid options worth considering, which could theoretically combine the benefits of the pipelines tested. The narrow band and NNS methods could be applied to SDF evaluation during the 3D texture construction in the simple pipeline, which may combine the complex pipeline's fast SDF evaluation with the simple pipeline's fast ray traversal, albeit at the cost of high memory overhead. Alternatively, SDF discretisation could be avoided and sphere tracing performed analytically within the generated AABBs, still utilising the narrow band and NNS methods. This would remove most of the memory overhead, but the effect on framerates would need to be studied.

List of References

Aaltonen, S. (2018) 'GPU-based clay simulation and ray tracing tech in Claybook' *Game Developers Conference*, San Francisco (United States), 19-23 March. Available at: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf (Accessed: 10 April 2025).

Adobe (2025) *Photoshop* [Computer program]. Available at: https://www.adobe.com/uk/products/photoshop (Accessed: 29 April 2025).

Akinci, G. *et al.* (2012) 'Parallel Surface Reconstruction for Particle-Based Fluids', *Computer Graphics Forum*, 31(6), pp. 1797-1809. Available at: https://doi.org/10.1111/j.1467-8659.2012.02096.x

Bálint, C., Valasek, G. and Gergó, L. (2019) 'Operations on Signed Distance Functions', *Acta Cybernetica*, 24(1), pp. 17-28. Available at: https://doi.org/10.14232/actacyb.24.1.2019.3

Blelloch, G. (1993) 'Prefix Sums and Their Applications', in J. Reif (ed.) *Synthesis of Parallel Algorithms*. San Francisco (United States): Morgan Kaufmann Publishers Inc., pp. 35-60.
Available at: https://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf (Accessed: 15 April 2025).

Crassin, C. *et al.* (2009) 'GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering', *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, Boston (United States), 27 February-1 March. New York (United States): Association for Computing Machinery, pp. 15-22. Available at: https://doi.org/10.1145/1507149.1507152

Evans, A. (2015) 'Learning from failure: A survey of promising, unconventional and mostly abandoned renderers for Dreams PS4, a geometrically dense, painterly UGC game', *SIGGRAPH '15: ACM SIGGRAPH 2015 Courses*, Los Angeles (United States), 9-13 August. Available at: https://advances.realtimerendering.com/s2015/AlexEvans_SIGGRAPH-2015-sml.pdf (Accessed: 10 April 2025).

Fisher, M. (2014) *Marching Cubes*. Available at: https://graphics.stanford.edu/~mdfisher/MarchingCubes.html (Accessed: 11 April 2025).

Green, S. (2010) *Particle Simulation using CUDA*. Available at: https://developer.download.nvidia.com/assets/cuda/files/particles.pdf (Accessed: 14 April 2025).

Hart, J. (1996) 'Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces', *The Visual Computer*, 12(10), pp. 527-545. Available at: https://doi.org/10.1007/s003710050084

Hoetzlein, R. (2014) ‘Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids’, *The GPU Technology Conference 2014*, San Jose (United States) 24-27 March. Available at: https://doczz.net/doc/8083119/fast-fixed-radius-nearest-neighbor-search-on-the-gpu (Accessed 14 April 2025).

Karlsson, B. (2025) *RenderDoc* [Computer program]. Available at: https://renderdoc.org/ (Accessed: 30 April 2025).

Kellomäki, T. (2012) ‘Water simulation methods for games: a comparison’, *MindTrek '12: Proceeding of the 16th International Academic MindTrek Conference*, Tampere (Finland), 3-5 October. New York (United States): Association for Computing Machinery, pp. 10-14. Available at: https://doi.org/10.1145/2393132.2393135

Koschier, D. *et al.* (2022) ‘A Survey on SPH Methods in Computer Graphics’, *Computer Graphics Forum*, 41(2), pp. 737-760. Available at: https://doi.org/10.1111/cgf.14508

Laine, S. and Karras, T. (2010) ‘Efficient Sparse Voxel Octrees’, *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, Washington D.C. (United States), 19-21 February. New York (United States): Association for Computing Machinery, pp. 55-63. Available at: https://doi.org/10.1145/1730804.1730814

Liu, B. *et al*. (2023) ‘Building a Real-Time System on GPUs for Simulation and Rendering of Realistic 3D Liquid in Video Games’, *SIGGRAPH '23: ACM SIGGRAPH 2023 Courses*, Los Angeles (United States), 6-10 August. New York (United States): Association for Computing Machinery. Available at: https://doi.org/10.1145/3587423.3595537

Lorensen, W. and Cline, H. (1987) ‘Marching cubes: A high resolution 3D surface construction algorithm’, *ACM SIGGRAPH Computer Graphics*, 21(4), pp. 163-169. Available at: https://doi.org/10.1145/37402.37422

Malan, H. (2022) ‘Rendering Water in Horizon Forbidden West’, *SIGGRAPH '22: ACM SIGGRAPH 2022 Courses*, Vancouver (Canada), 8-11 August. Available at: https://advances.realtimerendering.com/s2022/SIGGRAPH2022-Advances-Water-Malan.pdf (Accessed: 1 May 2025).

Merrill, D. and Garland, M. (2016) *Single-pass Parallel Prefix Scan with Decoupled Look-back*. NVIDIA Research paper nvr-2016-002. Available at: https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back (Accessed: 15 April 2025).

Microsoft (2021) *Direct3D 12 graphics*. Available at: https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (Accessed: 30 April 2025).

Microsoft (2023) *DirectX Raytracing (DXR) Functional Spec*. Available at: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing (Accessed: 30 April 2025).

Microsoft (2025) *Microsoft Excel* [Computer program]. Available at: https://www.microsoft.com/en-us/microsoft-365/excel (Accessed 29 April 2025).

Müller, M., Charypar, D. and Gross, M. (2003) 'Particle-based fluid simulation for interactive applications', *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. San Diego (United States), 26-27 July. Goslar (Germany): Eurographics Association, pp. 154-159. Available at: https://matthias-research.github.io/pages/publications/sca03.pdf (Accessed: 8 April 2025).

Nishidate, Y. and Fujishiro, I. (2024) 'Efficient Particle-Based Fluid Surface Reconstruction Using Mesh Shaders and Bidirectional Two-Level Grids', *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 7(1), pp. 1-14. Available at: https://doi.org/10.1145/3651285

NVIDIA Corporation (2025a) *CUDA C++ Programming Guide*. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accessed: 8 April 2025).

NVIDIA Corporation (2055b) *NVIDIA Nsight Graphics* [Computer program]. Available at: https://developer.nvidia.com/nsight-graphics (Accessed: 30 April 2025).

NVIDIA Corporation (2025c) *NVIDIA Nsight Perf SDK* [Computer program]. Available at: https://developer.nvidia.com/nsight-perf-sdk (Accessed: 28 April 2025).

Quilez, I. (2015) *normals for an SDF*. Available at: https://iquilezles.org/articles/normalsSDF/ (Accessed: 29 April 2025).

Quispe, F.I. and Paiva, A. (2022) 'Counting Particles: a simple and fast surface reconstruction method for particle-based fluids', *2022 35th SIBGRAPI Conference on Graphics, Patterns and Images*, Natal (Brazil), 24-27 October. IEEE, pp. 145-149. Available at: https://doi.org/10.1109/SIBGRAPI55357.2022.9991770

Smith, T. (2024) *GPUPrefixSums* [Computer program]. Available at: https://github.com/b0nes164/GPUPrefixSums (Accessed: 24 April 2025).

Söderlund, H., Evans A. and Akenine-Möller, T. (2022) 'Ray Tracing of Signed Distance Function Grids', *Journal of Computer Graphics Techniques*, 11(3), pp. 94-113. Available at: https://www.jcgt.org/published/0011/03/06 (Accessed: 10 April 2025).

Wikipedia (2024) *Scalar field*. Available at: https://en.wikipedia.org/wiki/Scalar_field (Accessed 9 April 2025).

Yang, W. and Gao, C. (2020) 'A completely parallel surface reconstruction method for particle-based fluids', *The Visual Computer*, 36, pp. 2313–2325. Available at: https://doi.org/10.1007/s00371-020-01898-2

Zhang, H. (2011) *Ray casting technique used in computer graphics, each pixel decides one casting ray*. Available at: https://www.researchgate.net/figure/a-Ray-casting-technique-used-in-computer-graphics-each-pixel-decides-one-casting-ray_fig1_252241877 (Accessed: 27 April 2025).

Zhang, J. *et al*. (2023) 'Developing Complex Geometry Isosurface Reconstruction Tool for Smoothed Particle Hydrodynamics Simulations', *ICIGP '23: Proceedings of the 2023 6th International Conference on Image and Graphics Processing*. Chongqing (China), 6-8 January. New York (United States): Association for Computing Machinery, pp. 58-63. Available at: https://doi.org/10.1145/3582649.3582675

Zhu, Y. and Bridson, R. (2005) 'Animating sand as a fluid', *ACM Transactions on Graphics*, 24(3), pp. 965-972. Available at: https://doi.org/10.1145/1073204.1073298

Appendices

Appendix A: Full Test Results Data

Performance Metrics - Test System #1									
Range	**Time (ns)**	**Warp Activity (%)**	**ALU Throughput (%)**	**LSU Throughput (%)**	**Warp Stall - Registers (%)**	**Warp Stall - Warp (%)**	**L2 Cache Hits (%)**	**RT Core Throughput (%)**	**SM Throughput (%)**
Variables Tested: Default, Pipeline: Complex									
Grid Construction	6237.037037	0.788533	0.155711	0.261189	0	0	94.029468	0	0.261189
Surface Detection	149083.7963	0.035292	0.033709	0.019935	0	0	83.600793	0	0.033721
AABB Creation	7964.814815	2.456378	0.506339	0.367354	0	0	99.90467	0	0.751129
BVH Construction	27340.74074	3.113269	1.670583	3.159464	0	0	83.471847	0	4.174507
Particle Reordering	75444.90741	0.502858	0.05313	0.135964	0	0.009005	92.623924	0	0.135964
Brick Pool	705437.963	63.730493	59.360626	62.442301	22.731803	22.731815	98.857401	0	70.533087
Ray Tracing	513647.6852	49.49426	42.924801	12.365334	21.961195	0	95.875052	65.550695	48.385763
Total/Peak	**1485156.944**	**63.730493**	**59.360626**	**62.442301**	**22.731803**	**22.731815**	**99.90467**	**65.550695**	**70.533087**
Variables Tested: Default, Pipeline: Naïve									
Ray Tracing	8469499.074	45.573255	27.084532	87.578314	23.541342	0	80.502149	3.507247	87.578314
Variables Tested: Default, Pipeline: Simple									
Simple Texture	17492981.94	62.842125	26.892188	97.712233	0	1.139641	70.739106	0	97.712233
Ray Tracing	432836.1111	50.535546	34.425956	9.078668	20.239785	0	90.353435	69.574179	35.728237
Total/Peak	**17925818.05**	**62.842125**	**34.425956**	**97.712233**	**20.239785**	**1.139641**	**90.353435**	**69.574179**	**97.712233**
Variables Tested: Particles - 1, Pipeline: Complex									
Grid Construction	5657.87037	0.715379	0.077151	0.236663	0	0	97.116271	0	0.236663
Surface Detection	141562.5	0.008002	0.005444	0.004786	0	0	87.58977	0	0.005911
AABB Creation	6412.962963	0.401118	0.044419	0.052642	0	0	98.239796	0	0.100859
BVH Construction	21337.5	0.220499	0.114318	0.222627	0	0	81.498371	0	0.222627
Particle Reordering	76575	0.472882	0.055619	0.135298	0	0.009389	82.244633	0	0.140682
Brick Pool	27615.74074	52.297447	49.048671	43.122667	8.261278	8.261593	93.743222	0	49.049268
Ray Tracing	281659.2593	46.841476	44.886649	11.090215	18.183337	0	96.896774	79.599883	45.169948
Total/Peak	**560820.8334**	**52.297447**	**49.048671**	**43.122667**	**18.183337**	**8.261593**	**98.239796**	**79.599883**	**49.049268**
Variables Tested: Particles - 1, Pipeline: Naïve									
Ray Tracing	408668.9815	50.346308	35.239217	10.800968	20.185157	0	95.072043	71.572209	36.191348
Variables Tested: Particles - 1, Pipeline: Simple									
Simple Texture	291104.6296	42.35862	29.985786	27.259806	0	0	99.514053	0	38.849744
Ray Tracing	409937.037	50.545256	33.683567	9.277948	19.993408	0	89.100624	71.07965	33.989169
Total/Peak	**701041.6666**	**50.545256**	**33.683567**	**27.259806**	**19.993408**	**0**	**99.514053**	**71.07965**	**38.849744**
Variables Tested: Particles - 27, Pipeline: Complex									
Grid Construction	6262.037037	0.648065	0.079026	0.226321	0	0	97.09731	0	0.226321
Surface Detection	152865.2778	0.050329	0.058906	0.026934	0	0	84.515941	0	0.058906
AABB Creation	7929.62963	1.112774	0.247676	0.184628	0	0	96.600224	0	0.375874
BVH Construction	24857.87037	1.260314	0.770266	1.457261	0	0	80.92068	0	1.457261
Particle Reordering	71455.55556	0.470588	0.053774	0.130812	0	0.009021	90.657316	0	0.136022
Brick Pool	209838.8889	61.9159	58.50487	53.983797	21.481639	21.481665	99.502846	0	60.378399
Ray Tracing	474312.5	49.822161	40.878342	11.443647	21.214891	0	96.915296	68.168743	43.975398
Total/Peak	**947521.7593**	**61.9159**	**58.50487**	**53.983797**	**21.481639**	**21.481665**	**99.502846**	**68.168743**	**60.378399**
Variables Tested: Particles - 27, Pipeline: Naïve									
Ray Tracing	660985.6481	49.486314	40.404366	64.59451	21.761191	0	93.92951	44.553665	70.806153
Variables Tested: Particles - 27, Pipeline: Simple									
Simple Texture	1526053.241	60.558659	28.205833	86.706821	0	0	96.285284	0	86.706821
Ray Tracing	417494.4444	50.572599	33.996415	9.129895	20.230077	0	88.337313	69.954521	34.86855
Total/Peak	**1943547.685**	**60.558659**	**33.996415**	**86.706821**	**20.230077**	**0**	**96.285284**	**69.954521**	**86.706821**
Variables Tested: Particles - 1000, Pipeline: Complex									
Grid Construction	7482.407407	1.251962	0.316939	0.321832	0	0	95.789328	0	0.416112
Surface Detection	144194.4444	0.121631	0.141206	0.061896	0	0	75.34718	0	0.141295
AABB Creation	8087.5	5.62221	1.1706	0.839149	0	0	98.533389	0	1.719068
BVH Construction	55676.38889	7.228895	3.41733	6.589298	0	0	88.278819	0	17.51943
Particle Reordering	75954.62963	0.653869	0.050909	0.143396	0	0.008601	93.487026	0	0.143396
Brick Pool	1895555.093	64.504912	57.742546	61.549551	23.291345	23.29135	94.427909	0	69.919464
Ray Tracing	1721797.685	52.729189	34.969264	20.673131	25.310598	0	96.982452	65.967214	48.877418
Total/Peak	**3908748.148**	**64.504912**	**57.742546**	**61.549551**	**25.310598**	**23.29135**	**98.533389**	**65.967214**	**69.919464**
Variables Tested: Particles - 1000, Pipeline: Naïve									
Ray Tracing	33949941.67	46.640887	26.980019	92.010717	24.858625	0	69.85824	0.893945	92.010717
Variables Tested: Particles - 1000, Pipeline: Simple									
Simple Texture	50732504.63	63.001604	26.825329	98.537334	0	13.386465	47.861944	0	98.537334
Ray Tracing	451154.1667	50.552791	34.664119	8.929611	20.427287	0	90.017035	68.34152	36.498035
Total/Peak	**51183658.8**	**63.001604**	**34.664119**	**98.537334**	**20.427287**	**13.386465**	**90.017035**	**68.34152**	**98.537334**
Variables Tested: Particles - 10648, Pipeline: Complex									
Grid Construction	7297.222222	8.56252	1.011908	1.109236	0	0	99.420878	0	1.481672
Surface Detection	134810.6481	0.12974	0.138102	0.075238	0	0	75.260758	0	0.138185
AABB Creation	9111.111111	9.105273	1.738756	1.238158	0	0.284524	99.462984	0	2.538692
BVH Construction	135113.4259	13.781562	5.119693	9.553067	2.310625	0	96.769672	0	61.567379
Particle Reordering	77666.66667	2.844125	0.068008	0.377027	0	0.008414	97.568467	0	0.377027
Brick Pool	8153308.333	65.523935	50.491543	75.279878	24.63761	24.637611	85.273025	0	77.096565
Ray Tracing	1126684.259	52.266694	31.968115	10.874973	25.427796	0	95.731018	52.299774	36.826729
Total/Peak	**9643991.666**	**65.523935**	**50.491543**	**75.279878**	**25.427796**	**24.637611**	**99.462984**	**52.299774**	**77.096565**
Variables Tested: Particles - 10648, Pipeline: Naïve									
Ray Tracing	587565960.2	46.460867	20.926635	72.685628	23.715689	0	96.553721	0.050922	72.685628
Variables Tested: Particles - 10648, Pipeline: Simple									
Simple Texture	563190277.3	66.216256	26.710675	98.576083	0	45.071682	84.009664	0	98.576083
Ray Tracing	480479.6296	50.680047	34.763551	8.728764	20.538241	0	89.491678	66.797604	37.137457
Total/Peak	**563670756.9**	**66.216256**	**34.763551**	**98.576083**	**20.538241**	**45.071682**	**89.491678**	**66.797604**	**98.576083**
Variables Tested: Scene - Grid, Pipeline: Complex									
Grid Construction	7887.962963	0.82492	0.155099	0.259813	0	0	76.364986	0	0.259813
Surface Detection	142702.3148	0.028501	0.023758	0.016135	0	0	70.251576	0	0.023884
AABB Creation	8094.444444	2.914558	0.616367	0.435082	0	0	99.189478	0	0.893618
BVH Construction	23054.16667	3.850253	2.348694	4.408555	0	0	80.700897	0	4.408555
Particle Reordering	72069.90741	0.487108	0.05363	0.137269	0	0.008953	84.610544	0	0.137269
Brick Pool	838302.7778	63.99495	60.258812	59.786316	22.781498	22.781504	96.890124	0	69.227454
Ray Tracing	514478.7037	49.025081	43.162565	11.772433	22.15512	0	95.552416	63.720967	47.258671
Total/Peak	**1606590.278**	**63.99495**	**60.258812**	**59.786316**	**22.781498**	**22.781504**	**99.189478**	**63.720967**	**69.227454**
Variables Tested: Scene - Grid, Pipeline: Naïve									
Ray Tracing	9927418.519	48.138404	28.845353	94.195508	25.805112	0	76.774228	3.197772	94.195508
Variables Tested: Scene - Grid, Pipeline: Simple									
Simple Texture	17479548.61	62.866239	26.909566	97.775375	0	1.07465	69.640882	0	97.775375
Ray Tracing	468410.1852	50.875732	33.381076	9.028747	20.64497	0	88.57129	67.966227	34.912553
Total/Peak	**17947958.8**	**62.866239**	**33.381076**	**97.775375**	**20.64497**	**1.07465**	**88.57129**	**67.966227**	**97.775375**

Variables Tested: Scene - Random, Pipeline: Complex									
Grid Construction	6440.277778	0.806657	0.186444	0.269227	0	0	96.78012	0	0.269227
Surface Detection	132894.9074	0.326991	0.288231	0.254346	0	0	87.816535	0	0.302666
AABB Creation	13168.51852	17.965622	2.660473	1.886049	0	1.914288	99.664323	0	3.836973
BVH Construction	184128.7037	24.636792	8.938488	16.378698	0.302595	0	98.588055	0	122.817506
Particle Reordering	74125.46296	0.51319	0.051875	0.132745	0	0.008873	88.851597	0	0.132745
Brick Pool	4106683.796	63.882187	51.236855	46.874737	22.713102	22.713103	91.454794	0	52.102301
Ray Tracing	4100215.741	53.174166	16.460249	6.26563	26.75021	0	91.776462	69.843632	18.870744
Total/Peak	**8617657.407**	**63.882187**	**51.236855**	**46.874737**	**26.75021**	**22.713103**	**99.664323**	**69.843632**	**122.817506**
Variables Tested: Scene - Random, Pipeline: Naïve									
Ray Tracing	39453722.69	43.855391	27.366853	93.087072	26.920342	0	64.802021	0.783096	93.087072
Variables Tested: Scene - Random, Pipeline: Simple									
Simple Texture	17544278.24	62.872608	26.912794	97.787105	0	1.064484	68.514101	0	97.787105
Ray Tracing	589718.9815	49.758116	31.871296	7.15151	22.738732	0	89.808501	53.700295	36.747344
Total/Peak	**18133997.22**	**62.872608**	**31.871296**	**97.787105**	**22.738732**	**1.064484**	**89.808501**	**53.700295**	**97.787105**
Variables Tested: Screen Res - 720p, Pipeline: Complex									
Grid Construction	6231.944444	0.759042	0.159799	0.267658	0	0	94.297179	0	0.267658
Surface Detection	147550	0.034393	0.034382	0.02035	0	0	71.140706	0	0.034386
AABB Creation	7975.925926	2.374361	0.512901	0.372055	0	0	97.304204	0	0.760751
BVH Construction	20043.51852	2.641286	1.845496	3.490115	0	0	90.964056	0	3.490115
Particle Reordering	74368.05556	0.446658	0.051736	0.132396	0	0.008341	79.415587	0	0.132396
Brick Pool	706215.7407	63.695624	59.418508	62.496043	22.759181	22.759186	93.184934	0	70.599821
Ray Tracing	257961.1111	43.011057	39.997327	11.344031	19.397342	0	99.130372	57.236874	45.077667
Total/Peak	**1220346.296**	**63.695624**	**59.418508**	**62.496043**	**22.759181**	**22.759186**	**99.130372**	**57.236874**	**70.599821**
Variables Tested: Screen Res - 720p, Pipeline: Naïve									
Ray Tracing	4592223.611	38.237865	23.430727	76.149255	19.813093	0	64.203833	2.787259	76.149255
Variables Tested: Screen Res - 720p, Pipeline: Simple									
Simple Texture	17579160.19	62.861233	26.90644	97.764018	0	3.505099	65.509525	0	97.764018
Ray Tracing	207581.9444	48.169822	32.447623	8.519031	19.085184	0	87.886633	64.671772	33.744479
Total/Peak	**17786742.13**	**62.861233**	**32.447623**	**97.764018**	**19.085184**	**3.505099**	**87.886633**	**64.671772**	**97.764018**
Variables Tested: Screen Res - 1440p, Pipeline: Complex									
Grid Construction	6504.166667	0.802215	0.154091	0.256787	0	0	94.523065	0	0.256787
Surface Detection	157153.7037	0.039986	0.036867	0.021818	0	0	88.535399	0	0.036877
AABB Creation	7997.685185	2.478743	0.508788	0.369204	0	0	99.975168	0	0.754835
BVH Construction	36799.53704	3.120801	1.626993	3.076512	0	0	84.892514	0	5.391313
Particle Reordering	72529.16667	0.5166	0.051456	0.131699	0	0.009069	94.038496	0	0.131699
Brick Pool	708012.5	63.692465	59.308417	62.406459	22.644621	22.64463	98.432447	0	70.479467
Ray Tracing	864807.4074	49.939698	42.353543	12.36138	22.550195	0	94.96463	66.626603	46.508106
Total/Peak	**1853804.167**	**63.692465**	**59.308417**	**62.406459**	**22.644621**	**22.64463**	**99.975168**	**66.626603**	**70.479467**
Variables Tested: Screen Res - 1440p, Pipeline: Naïve									
Ray Tracing	13486478.7	47.987698	28.744336	92.512946	24.865553	0	81.38203	3.907399	92.512946
Variables Tested: Screen Res - 1440p, Pipeline: Simple									
Simple Texture	17567253.24	62.866818	26.907892	97.769291	0	1.105854	68.753588	0	97.769291
Ray Tracing	743243.5185	51.305656	35.49804	9.3338	20.500704	0	91.509433	71.081653	36.718652
Total/Peak	**18310496.76**	**62.866818**	**35.49804**	**97.769291**	**20.500704**	**1.105854**	**91.509433**	**71.081653**	**97.769291**
Variables Tested: Screen Res - 4k, Pipeline: Complex									
Grid Construction	6428.703704	0.795022	0.155415	0.262409	0	0	98.021731	0	0.262409
Surface Detection	136018.5185	0.038004	0.035617	0.021086	0	0	89.499546	0	0.035629
AABB Creation	8012.5	2.442337	0.513693	0.372784	0	0	94.373679	0	0.762157
BVH Construction	31833.7963	3.091951	1.637502	3.096036	0	0	80.383314	0	4.779517
Particle Reordering	80328.7037	0.532004	0.052062	0.133194	0	0.009148	90.791308	0	0.133194
Brick Pool	706595.8333	63.667755	59.59731	62.712084	22.547659	22.547672	97.728693	0	70.825705
Ray Tracing	874999.537	50.170279	42.169444	12.161285	22.135759	0	95.574646	66.357487	46.675248
Total/Peak	**1844217.593**	**63.667755**	**59.59731**	**62.712084**	**22.547659**	**22.547672**	**98.021731**	**66.357487**	**70.825705**
Variables Tested: Screen Res - 4k, Pipeline: Naïve									
Ray Tracing	13497206.94	48.530688	28.84467	92.841466	24.810501	0	80.303584	3.933973	92.841466
Variables Tested: Screen Res - 4k, Pipeline: Simple									
Simple Texture	17465682.41	62.861818	26.90513	97.759256	0	3.57858	69.87101	0	97.759256
Ray Tracing	739080.0926	51.326466	35.372818	9.299438	20.639813	0	90.77539	71.107622	36.611378
Total/Peak	**18204762.5**	**62.861818**	**35.372818**	**97.759256**	**20.639813**	**3.57858**	**90.77539**	**71.107622**	**97.759256**
Variables Tested: Texture Res - 128, Pipeline: Complex									
Grid Construction	6312.5	0.769594	0.15868	0.266505	0	0	91.222688	0	0.266505
Surface Detection	149282.8704	0.036362	0.035764	0.02112	0	0	64.017986	0	0.03577
AABB Creation	6543.518519	0.441512	0.081904	0.073371	0	0	93.937897	0	0.145385
BVH Construction	24632.87037	0.435664	0.272911	0.512147	0	0	77.562347	0	0.642549
Particle Reordering	71435.18519	0.497407	0.054902	0.140505	0	0.009198	85.377897	0	0.140505
Brick Pool	108898.6111	59.70888	48.373044	50.847714	20.730003	20.730056	95.492302	0	57.467789
Ray Tracing	481552.3148	49.578851	42.81079	14.118444	21.610012	0	97.352058	68.769236	50.605927
Total/Peak	**848657.8704**	**59.70888**	**48.373044**	**50.847714**	**21.610012**	**20.730056**	**97.352058**	**68.769236**	**57.467789**
Variables Tested: Texture Res - 128, Pipeline: Simple									
Simple Texture	2124136.574	62.260942	26.599134	96.647428	0	2.448957	80.829164	0	96.647428
Ray Tracing	415712.037	50.471794	34.974047	9.214747	20.282264	0	95.458595	70.578609	36.304157
Total/Peak	**2539848.611**	**62.260942**	**34.974047**	**96.647428**	**20.282264**	**2.448957**	**95.458595**	**70.578609**	**96.647428**
Variables Tested: Texture Res - 512, Pipeline: Complex									
Grid Construction	6358.796296	0.786742	0.154999	0.260267	0	0	92.349577	0	0.260267
Surface Detection	133511.5741	0.033235	0.031028	0.018632	0	0	77.385748	0	0.031064
AABB Creation	11969.44444	17.87613	2.73382	1.936006	0	1.329653	99.52207	0	3.950587
BVH Construction	63647.68519	22.177006	8.316026	15.320533	0.318314	0	88.925331	0	35.466418
Particle Reordering	72287.03704	0.532416	0.053652	0.13729	0	0.009243	82.888543	0	0.13729
Brick Pool	5483974.537	63.505736	61.431303	64.621283	19.784663	19.784664	91.061992	0	72.98914
Ray Tracing	738798.6111	51.040836	40.298862	13.817638	23.313154	0	86.930924	61.176997	45.003205
Total/Peak	**6510547.685**	**63.505736**	**61.431303**	**64.621283**	**23.313154**	**19.784664**	**99.52207**	**61.176997**	**72.98914**
Variables Tested: Texture Res - 512, Pipeline: Simple									
Simple Texture	143453974.5	62.910939	26.942407	97.894701	0	0.938316	63.058464	0	97.894701
Ray Tracing	533638.8889	51.195038	29.700309	7.837415	20.906382	0	64.232486	59.986427	30.821722
Total/Peak	**143987613.4**	**62.910939**	**29.700309**	**97.894701**	**20.906382**	**0.938316**	**64.232486**	**59.986427**	**97.894701**
Variables Tested: Texture Res - 768, Pipeline: Complex									
Grid Construction	6293.055556	0.794645	0.155207	0.259447	0	0	88.469365	0	0.259447
Surface Detection	184436.1111	0.031117	0.028688	0.017111	0	0	66.628379	0	0.028727
AABB Creation	23122.68519	30.976209	4.75195	3.357774	0	14.629837	99.629079	0	6.65891
BVH Construction	166249.537	42.159894	16.61576	30.752864	0.125071	1.242431	91.361781	0	108.083888
Particle Reordering	83067.12963	0.521119	0.051795	0.132574	0	0.009142	81.642371	0	0.132574
Brick Pool	18306517.13	63.362688	61.855825	65.12422	15.697368	15.697368	89.516036	0	73.536816
Ray Tracing	1026808.333	51.176893	37.342965	13.909019	23.940206	0	79.76391	56.495825	41.952152
Total/Peak	**19796493.98**	**63.362688**	**61.855825**	**65.12422**	**23.940206**	**15.697368**	**99.629079**	**56.495825**	**108.083888**
Variables Tested: Texture Res - 768, Pipeline: Simple									
Simple Texture	490028787.5	62.918124	26.946838	97.910802	0	1.085448	62.340116	0	97.910802
Ray Tracing	728990.7407	51.984007	21.590996	5.697316	22.497091	0	46.977923	43.635706	22.40772
Total/Peak	**490757778.2**	**62.918124**	**26.946838**	**97.910802**	**22.497091**	**1.085448**	**62.340116**	**43.635706**	**97.910802**

Variables Tested: View Dist - 1, Pipeline: Naïve									
Ray Tracing	16606063.43	49.473573	28.935645	95.774787	25.375056	0	74.179661	2.391287	95.774787
Variables Tested: View Dist - 1, Pipeline: Simple									
Simple Texture	17739166.2	62.873364	26.912967	97.787731	0	1.058118	68.52536	0	97.787731
Ray Tracing	600166.6667	52.305014	31.139603	9.078262	21.134632	0	91.243268	64.701294	33.354514
Total/Peak	**18339332.87**	**62.873364**	**31.139603**	**97.787731**	**21.134632**	**1.058118**	**91.243268**	**64.701294**	**97.787731**
Variables Tested: View Dist - 2, Pipeline: Complex									
Grid Construction	6361.574074	0.780885	0.158467	0.264183	0	0	88.623356	0	0.264183
Surface Detection	152757.4074	0.035926	0.034463	0.020418	0	0	63.308544	0	0.034474
AABB Creation	8037.037037	2.477604	0.510455	0.370387	0	0	97.911301	0	0.757324
BVH Construction	32016.2037	2.952447	1.728733	3.268687	0	0	79.80024	0	4.930938
Particle Reordering	75091.2037	0.481726	0.051783	0.132497	0	0.008733	82.064708	0	0.132497
Brick Pool	708046.2963	63.711332	59.356186	62.451258	22.746053	22.746063	96.956171	0	70.533427
Ray Tracing	380567.1296	48.327253	44.644371	11.584058	20.509879	0	94.70884	68.999058	47.82261
Total/Peak	**1362876.852**	**63.711332**	**59.356186**	**62.451258**	**22.746053**	**22.746063**	**97.911301**	**68.999058**	**70.533427**
Variables Tested: View Dist - 2, Pipeline: Naïve									
Ray Tracing	5592580.556	41.345832	25.133179	79.044056	21.102065	0	79.306785	4.462013	79.044056
Variables Tested: View Dist - 2, Pipeline: Simple									
Simple Texture	17451394.91	62.848433	26.892225	97.712367	0	1.152945	70.221616	0	97.712367
Ray Tracing	351608.7963	48.976198	37.282837	8.982072	19.309813	0	88.053708	73.036575	38.062863
Total/Peak	**17803003.71**	**62.848433**	**37.282837**	**97.712367**	**19.309813**	**1.152945**	**88.053708**	**73.036575**	**97.712367**
Variables Tested: View Dist - 2.5, Pipeline: Complex									
Grid Construction	6329.166667	0.793563	0.156426	0.263332	0	0	95.794953	0	0.263332
Surface Detection	149133.3333	0.036542	0.035735	0.021133	0	0	75.824873	0	0.035735
AABB Creation	8004.166667	2.398082	0.514076	0.372973	0	0	99.110971	0	0.76264
BVH Construction	27230.55556	2.850791	1.761758	3.331683	0	0	82.945264	0	4.252331
Particle Reordering	73528.24074	0.492926	0.052913	0.135389	0	0.0089	87.109946	0	0.135389
Brick Pool	708847.6852	63.718489	59.357646	62.421254	22.669021	22.669026	95.360238	0	70.520491
Ray Tracing	331173.6111	47.304134	45.797871	11.079938	19.294028	0	95.224321	72.677906	49.113696
Total/Peak	**1304246.759**	**63.718489**	**59.357646**	**62.421254**	**22.669021**	**22.669026**	**99.110971**	**72.677906**	**70.520491**
Variables Tested: View Dist - 2.5, Pipeline: Naïve									
Ray Tracing	4419827.778	36.080035	22.660114	69.000701	17.918189	0	82.649566	5.355241	69.000701
Variables Tested: View Dist - 2.5, Pipeline: Simple									
Simple Texture	17625757.87	62.874147	26.91316	97.788435	0	1.047335	68.950109	0	97.788435
Ray Tracing	313103.7037	48.106403	39.328529	8.844873	18.457562	0	84.935634	75.216046	39.753706
Total/Peak	**17938861.57**	**62.874147**	**39.328529**	**97.788435**	**18.457562**	**1.047335**	**84.935634**	**75.216046**	**97.788435**

Performance Metrics - Test System #2

Range	Time (ns)	Warp Activity (%)	ALU Throughput (%)	LSU Throughput (%)	Warp Stall - Registers (%)	Warp Stall - Warp (%)	L2 Cache Hits (%)	RT Core Throughput (%)	SM Throughput (%)
Variables Tested: Default, Pipeline: Complex									
Grid Construction	6741.666667	0.132728	0.02735	0.046931	0	0	89.867696	0	0.046931
Surface Detection	185238.4259	0.006192	0.00614	0.003648	0	0	67.281146	0	0.006141
AABB Creation	8709.259259	0.411861	0.091922	0.066669	0	0	93.928827	0	0.136316
BVH Construction	20905.09259	0.433286	0.335134	0.633727	0	0	92.930705	0	0.633727
Particle Reordering	81493.51852	0.062024	0.009746	0.024941	0	0	68.523926	0	0.024941
Brick Pool	165326.8519	61.277063	50.786318	53.405248	21.663574	21.663576	95.326229	0	60.343559
Ray Tracing	165811.1111	32.0063	26.737573	7.661489	12.747614	0	98.680133	39.873487	29.12644
Total/Peak	**634225.9259**	**61.277063**	**50.786318**	**53.405248**	**21.663574**	**21.663576**	**98.680133**	**39.873487**	**60.343559**
Variables Tested: Default, Pipeline: Naïve									
Ray Tracing	2967327.315	24.733115	15.120837	48.914115	11.225898	0	74.693049	1.940349	48.914115
Variables Tested: Default, Pipeline: Simple									
Simple Texture	3453626.389	62.646772	26.808382	97.407723	0	4.564479	81.313035	0	97.407723
Ray Tracing	100376.8519	42.869526	28.936994	7.62481	15.995973	0	98.997201	58.376408	30.066055
Total/Peak	**3554003.241**	**62.646772**	**28.936994**	**97.407723**	**15.995973**	**4.564479**	**98.997201**	**58.376408**	**97.407723**

Performance Metrics - Test System #3

Range	Time (ns)	Warp Activity (%)	ALU Throughput (%)	LSU Throughput (%)	Warp Stall - Registers (%)	Warp Stall - Warp (%)	Warp Stall - GSM (%)	SM Throughput (%)
Variables Tested: Default, Pipeline: Complex								
Grid Construction	8435.648148	1.34527	0.234608	0.330306	0	0	0	0.379858
Surface Detection	460198.1481	0.023919	0.017838	0.010491	0	0	0	0.018474
AABB Creation	14206.01852	4.522018	0.374094	0.319675	0	0	0	0.999629
BVH Construction	54139.35185	3.190586	0.974402	1.79415	0	0	0	1.79415
Particle Reordering	248877.3148	0.292659	0.04326	0.060788	0	0.003215	0	0.138505
Brick Pool	1070002.315	95.295255	48.15706	61.410453	45.636768	45.699427	0	64.396721
Ray Tracing	1352258.796	41.835726	45.587415	15.618227	57.121458	0	35.853708	56.689957
Total/Peak	**3208117.593**	**95.295255**	**48.15706**	**61.410453**	**57.121458**	**45.699427**	**35.853708**	**64.396721**
Variables Tested: Default, Pipeline: Naïve								
Ray Tracing	10230038.89	37.059537	21.908719	71.505952	59.216929	0	31.328387	71.505952
Variables Tested: Default, Pipeline: Simple								
Simple Texture	28407244.44	97.338551	25.93661	94.406514	0	41.972527	0	94.406514
Ray Tracing	631274.0741	44.670526	39.494943	18.144485	42.216855	0	35.090133	45.748909
Total/Peak	**29038518.52**	**97.338551**	**39.494943**	**94.406514**	**42.216855**	**41.972527**	**35.090133**	**94.406514**

Memory Usage (Bytes)

		Narrow Band Grid							NNS				
Pipeline	Test Var	BlocksBuff	CellsBuf	SurfaceBlockInc	SurfaceCellIndicesBu	SurfaceCounts	SurfaceCounts	Total	OrderedPartic	GlobalIndexO	ScanBumpBuf	ThreadReduc	Total
Complex	Default	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Default	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Default	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Particles - 1	256	16384	256	16384	8	8	33296	36	16384	4	8	16432
Naïve	Particles - 1	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Particles - 1	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Particles - 27	256	16384	256	16384	8	8	33296	972	16384	4	8	17368
Naïve	Particles - 27	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Particles - 27	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Particles - 1000	256	16384	256	16384	8	8	33296	36000	16384	4	8	52396
Naïve	Particles - 1000	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Particles - 1000	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Particles - 10648	256	16384	256	16384	8	8	33296	383328	16384	4	8	399724
Naïve	Particles - 10648	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Particles - 10648	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Scene - Grid	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Scene - Grid	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Scene - Grid	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Scene - Random	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Scene - Random	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Scene - Random	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Screen Res - 720p	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Screen Res - 720p	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Screen Res - 720p	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Screen Res - 1440p	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Screen Res - 1440p	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Screen Res - 1440p	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Screen Res - 4k	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	Screen Res - 4k	0	0	0	0	0	0	0	0	0	0	0	0
Simple	Screen Res - 4k	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Texture Res - 128	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Simple	Texture Res - 128	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Texture Res - 512	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Simple	Texture Res - 512	0	0	0	0	0	0	0	0	0	0	0	0
Complex	Texture Res - 768	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Simple	Texture Res - 768	0	0	0	0	0	0	0	0	0	0	0	0
Complex	View Dist - 1	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	View Dist - 1	0	0	0	0	0	0	0	0	0	0	0	0
Simple	View Dist - 1	0	0	0	0	0	0	0	0	0	0	0	0
Complex	View Dist - 2	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	View Dist - 2	0	0	0	0	0	0	0	0	0	0	0	0
Simple	View Dist - 2	0	0	0	0	0	0	0	0	0	0	0	0
Complex	View Dist - 2.5	256	16384	256	16384	8	8	33296	12348	16384	4	8	28744
Naïve	View Dist - 2.5	0	0	0	0	0	0	0	0	0	0	0	0
Simple	View Dist - 2.5	0	0	0	0	0	0	0	0	0	0	0	0

AABBs/BLAS					3D Textures			
ComplexAABBE	SimpleAABBBuffer	ComplexBLAS	SimpleBLAS	Total	BrickPool	SimpleTexture	Total	Overall Total
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
5184	0	9472	0	14656	921464	0	921464	985848
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
43036	0	62361	0	105397	6955291	0	6955291	7111352
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
207590	0	292211	0	499801	21342626	0	21342626	21928119
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
333696	0	468224	0	801920	29439988	0	29439988	30674928
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
98112	0	139264	0	237376	13107008	0	13107008	13406424
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
710976	0	995328	0	2E+06	73727852	0	73727852	75496196
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
11842	0	18956	0	30798	1612704	0	1612704	1705542
0	24	0	1152	1176	0	4194304	4194304	4195480
758784	0	1062099	0	2E+06	77721469	0	77721469	79604392
0	24	0	1152	1176	0	268435456	268435456	268436632
2560896	0	3579276	0	6E+06	228848573	0	228848573	235050785
0	24	0	1152	1176	0	905969664	905969664	905970840
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608
94742	0	134540	0	229282	12940608	0	12940608	13231930
0	24	0	1152	1176	0	0	0	1176
0	24	0	1152	1176	0	33554432	33554432	33555608

Appendix B: GDPR Data Management Sign Off Form

Abertay University

GDPR Research Data Management
Data Sign Off Form

For undergraduate or postgraduate student projects supervised by an Abertay staff member.

This form MUST be included in the student's thesis/dissertation. Note that failure to do this will mean that the student's project cannot be assessed/examined.

Part 1: Supervisors to Complete

By signing this form, you are confirming that you have checked and verified your student's data according to the criteria stated below (e.g., raw data, completed questionnaires, superlab/Eprime output, transcriptions etc.)

Student Name:	George Fyles		
Student Number:	2100707		
Lead Supervisor Name:	Erin Hughes		
Lead Supervisor Signature	*Erin E Hughes*		
Project title:	An Optimised Pipeline for the Non-Polygonal Rendering of Particle-Based Fluid Simulations		
Study route:	PhD ☐	MbR ☐	MPhil ☐
	Undergraduate ☒	PhD by Publication ☐	

Part 2: Student to Complete

	Initial here to confirm 'Yes'
I confirm that I have handed over all manual records from my research project (e.g., consent forms, transcripts) to my supervisor for archiving/storage	G.F.
I confirm that I have handed over all digital records from my research project (e.g., recordings, data files) to my supervisor for archiving/storage	G.F.
I confirm that I no longer hold any digital records from my research project on any device other than the university network and the only data that I may retain is a copy of an anonymised data file(s) from my research	G.F.
I understand that, for undergraduate projects, my supervisor may delete manual/digital records of data if there is no foreseeable use for that data (with the exception of consent forms, which should be retained for 10 years)	G.F.

Student signature : 

Date: 06/05/2025