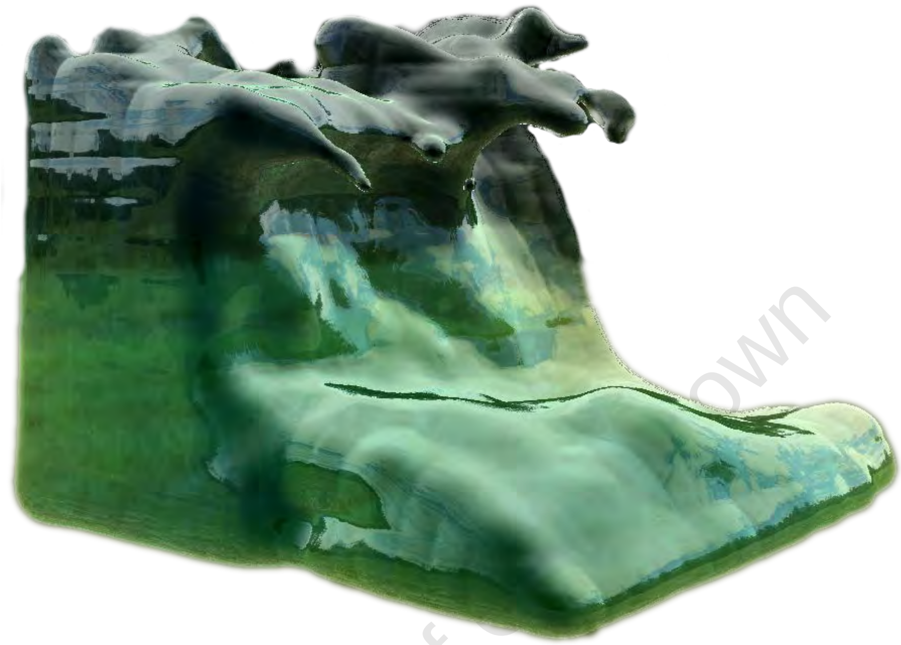
Interactive Simulation and Rendering of Fluids on Graphics Hardware



by

Shaun Silson

Thesis presented for the degree of

Master of Science

In the Department of Computer Science

University of Cape Town

June 2015

Supervised by:

Assoc Professor J. E. Gain

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the ACM Convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been attributed, and has been cited and referenced.
3. This thesis is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.
5. I acknowledge that copying someone else's work, or part of it, is wrong. I know the meaning of plagiarism and declare that all of the work in the thesis, save for that which is properly acknowledged, is my own.

   Signature ______________

Abstract

Computational fluid dynamics can be used to reproduce the complex motion of fluids for use in computer graphics, but the simulation and rendering are both highly computationally intensive. In the past performing these tasks on the CPU could take many minutes per frame, especially for large scale scenes at high levels of detail, which limited their usage to offline applications such as in film and media. However, using the massive parallelism of GPUs, it is nowadays possible to produce fluid visual effects in real time for interactive applications such as games. We present such an interactive simulation using the CUDA GPU computing environment and OpenGL graphics API.

Smoothed Particle Hydrodynamics (SPH) is a popular particle-based fluid simulation technique that has been shown to be well suited to acceleration on the GPU. Our work extends an existing GPU-based SPH implementation by incorporating rigid body interaction and rendering. Solid objects are represented using particles to accumulate hydrodynamic forces from surrounding fluid, while motion and collision handling are handled by the Bullet Physics library on the CPU. Our system demonstrates two-way coupling with multiple objects floating, displacing fluid and colliding with each other. For rendering we compare the performance and memory consumption of two approaches, splatting and raycasting, we also describe the visual characteristics of each.

In our evaluation we consider a target of between 24 and 30 fps to be sufficient for smooth interaction and aim to determine the performance impact of our new features. We begin by establishing a performance baseline and find that the original system runs smoothly up to 216,000 fluid particles but after introducing rendering this drops to 27,000 particles with the rendering taking up the majority of the frame time in both techniques. We find that the most significant limiting factor to splatting performance to be the onscreen area occupied by fluid while the raycasting performance is primarily determined by the resolution of the 3D texture used for sampling. Finally we find that performing solid interaction on the CPU is a viable approach that does not introduce significant overhead unless solid particles vastly outnumber fluid ones.

Contents

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is the use of numerical methods to simulate and analyze fluid flows and has applications in a wide range of fields, such as aerospace design, chemical engineering, oceanography and meteorology. This work focuses on the use of fluid simulation in computer graphics, where it is used to produce, using simulation, scenes which would be too tedious or difficult to produce using traditional animation. This allows more complex and visually rich scenes to be created. Although the term "fluid" technically refers to both liquid and gaseous materials, we will be focusing exclusively on liquids throughout this work. Nevertheless, similar techniques can be used to simulate other fluid-like phenomena such as clouds, smoke and fire.

While it does save time in animating, fluid simulation is inherently computationally expensive, which means that when it was first introduced in computer animation it's use was exclusively offline. This means setting up initial conditions for a scene, running the simulation program to generate motion and then checking the results afterwards. If problems are found then parameters are adjusted and the simulation process is run again. This cycle is repeated until the desired motion is achieved. Since rendering is also expensive, usually the fluid motion will be finalized with low quality draft or wireframe rendering and then final, high-quality rendering is performed in a separate pass. This type of workflow means long delays between setting up and getting results, and so fluid dynamics effects could only be used to generate pre-computed animations. While this is acceptable for movies where all animation is done beforehand, it does not work for interactive applications like games. In these case water effects are usually achieved using much simpler and cheaper to computes models like sinusoidal waves or heightmap fields. These are only able to capture simple surfaces and do not reproduce the splashing, swirling and vortices of real-world fluid dynamics.

Nevertheless, with the continuing increase in computing power, it is nowadays feasible to run full fluid simulations at interactive rates. A major contributing factor is the rise in popularity of powerful, inexpensive consumer hardware, specifically the Graphical Processing Unit (GPU). Initially graphics processors were dedicated single function hardware which accelerated only the most expensive part of rendering, the rasterization. Later generations of GPUs became fully programmable, allowing full access to the underlying computing power of the hardware. At first this programmable power could only be accessed using specialized graphics APIs, but soon toolkits were developed to allow general purpose code to run on the GPU.

This approach of using shaders to solve non-graphical problems became known as General Purpose GPU (GPGPU) computing, and it became popular for many high performance applications, including fluid simulation which is particularly well suited for parallelization since it involves repetitive application of simple instructions to large amounts of data. In 2007, NVIDIA released the CUDA toolkit which enabled GPGPU computing to be performed without having to work via the graphics API, which introduced the power of GPU computing to an even wider audience.

Although the basic data-processing problems of fluid simulation are solved with more horsepower, there are still other issues which need to be considered. Two important aspects of any fluid simulation are rendering and interaction with solid objects, both of which are relatively well solved problems offline but remain challenging to do in realtime. Solid-fluid coupling is still an open research problem and much of the research done has been CPU based, a large amount of effort if required to reproduce the same results on the GPU. As an example, one of the most popular products for computer generated fluid effects is Realflow [5] and it only ported it's offline fluid solver to the GPU as late as 2013.

1.1 Aims and Approach

Our aim is to produce a realistic looking fluid simulation containing solid objects that responds immediately to user interaction. While we have found in the literature that there are many works describing these individual aspects separately, we have not found any work that incorporates all of them into a single coherent system that could potentially be incorporated into interactive applications such as 3D games. This introduces several limitations on how many particles can be simulated in a scene in comparison to offline rendering applications such as film, both with regards to how much memory the particles occupy and how long it takes to process them. A major part of our work focuses on identifying and measuring these constraints especially with regards to rendering.

We extend the work of Hoetzlein [29], a GPU based implementation of Smoothed Particle Hydrodynamics (SPH) which is a popular particle-based method of simulating fluid. While Hoetzlein's system performs interactive fluid simulation it does not currently have rendering or solid-fluid coupling, so we will incorporate these two features and then evaluate the impact they have on performance. We will determine if interactive simulations are still feasible and at what scale. In the case of fluid rendering there are two different approaches, we implement both and compare their strengths and weaknesses. In the case of solid-fluid coupling there is a far broader range of considerations required to implement all aspects of physical realism. We will only implement the simplest case of rigid bodies defined in terms of particles, similar to the fluid particles. This will be a proof of concept upon which more complex physical models can later be built.

In order to evaluate our results we provide the following key objectives:

- The entire system must be capable of running between 24 and 30 frames per second. This is generally considered to be the minimum required framerate to maintain the illusion of smooth motion for the human visual system as opposed to a series of still images. The primary factor in overall system performance is the number of particles that can be simulated at this rate, therefore we will establish a baseline by finding the largest number of particles the existing system can simulate at this rate and then measuring the impact of introducing rendering and solids.
- Our rendering system must reproduce the key visual characteristics of liquid water as convincingly as possible. Since increased rendering quality usually comes at the expense of performance we expect there to be a tradeoff between these factors which we will attempt to quantify and find a balance for.
- Our solid objects must be able to float freely and move in a plausible manner in response to the fluid flowing around them. They must also exert appropriate forces back on the fluid by displacement and splashing.

1.2 Thesis Structure

Chapter 2 provides a background introduction to the fluid simulation topics covered in this thesis. We describe the two fundamental approaches to fluid simulation and providing a review of previous work in fluid simulation, originally as applied to offline animation (generally CPU based) and then moving on to interactive simulation using GPU acceleration. We conclude this chapter by exploring the various approaches to solid-fluid coupling.

The next three chapters introduce the theoretical aspects of each part of our final system. First chapter 3 describes how GPUs can use massive parallelism to accelerate computationally intensive tasks, and specifically describes how the CUDA programming environment can be used to harness this power. Secondly, chapter 4 begins by breaking down the Navier Stokes Equations, which provide the underlying mathematical framework for all fluid simulations, and then moves on to describe the details of Smoothed Particle Hydrodynamics which is a popular method used to implement these equations. Thirdly, chapter 5 explores the two general approaches to rendering, image order and object order, and covers the key optical properties of liquids which we aim to reproduce.

With the required theoretical grounding established, chapters 6 and 7 discuss the implementation of the simulation and rendering respectively which make up two distinct phases of the overall system. We describe how Hoetzlein's original system implements SPH in CUDA and how we incorporate solid objects using the Bullet physics engine to handle rigid body motion. Both our rendering solutions are implemented in OpenGL using CUDA's interop functionality to transfer data from the simulation to the rendering phase.

Finally chapter 8 presents the results and analyses of each aspect of the system to determine if our original aims have been met and chapter 9 presents conclusions and suggestions for possible future work.

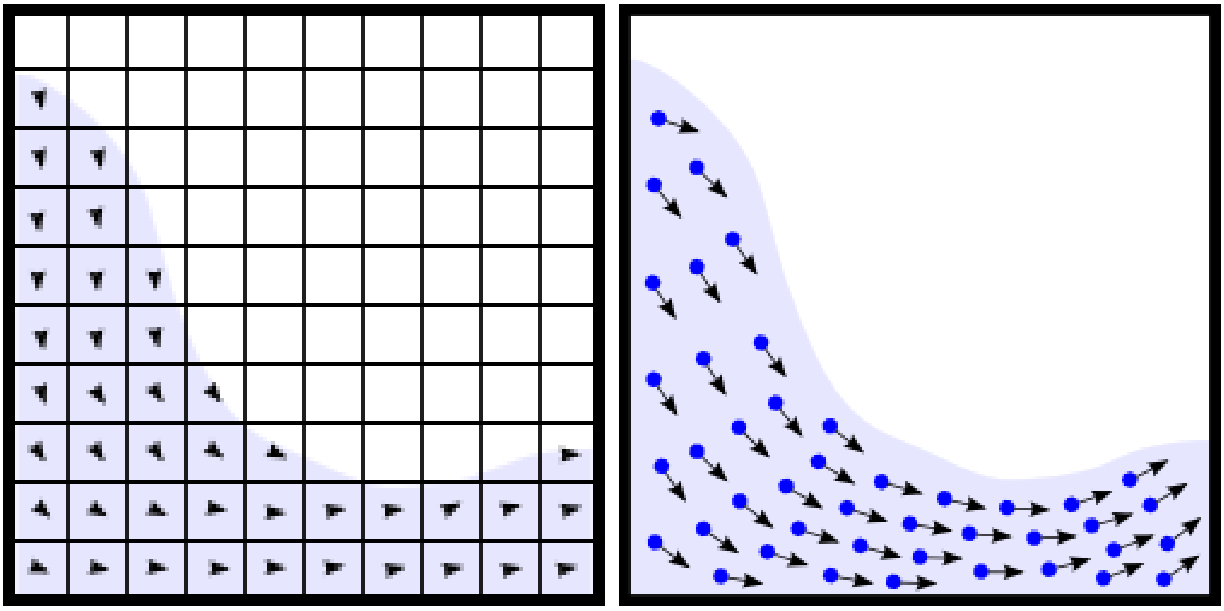
Chapter 2

Background and Previous Work

2.1 Fundamental Models of Fluid Simulation

Ultimately, all fluid simulations are based on the Navier Stokes equations which describe the physics of fluid in motion. This motion occurs due to the influence of three key forces: pressure, viscosity and external forces (for example wind or gravity). As a fluid system evolves in time the equations output a 2 or 3 dimensional field of values representing (at each time step) the position of the fluid in space. There are two different ways of representing these positions: *Eulerian* and *Lagrangian*. The Eulerian interpretation (figure 2.1a) describes a grid of cells fixed in space where each cell has a scalar pressure value and the fluid flows from cells with high to cells with lower pressure. The Lagrangian interpretation (figure2.1b) describes the fluid as a set of particles which exert forces on each other. Areas with many particles have higher pressure while areas with fewer particles have low, and again fluid moves along the pressure gradient. In chapter 4 we will describe in more detail the mathematical theory of Smoothed Particle Hydrodynamics (SPH) which is popular a Lagrangian technique, but first we will discuss the various trade-offs between the two approaches, highlighting where each might be more appropriate.

The Eulerian approach is well suited where accurate physical measurements need to be made at particular points, for example modelling a plane in a wind tunnel or water flowing along pipes. It also works well where a large-scale overview of a large area is required such as modelling ocean currents. The main issue with Eulerian formulations is that they have difficulty with free surfaces. This means that capturing visually interesting detail like splashes and spray, requires excessively high grid resolution which adversely affects performance and memory consumption. A benefit of the Lagrangian approach is that the particles naturally capture such fine detail, but a large amount of particles is required to cover a large area or volume. This trade-off between high levels of detail and large amounts of memory consumption can be addressed in several ways. Adaptive sampling can be used to increase grid resolution or particle density in particular regions, Adams et al [8] us a Lagrangian approach and split particles in regions of high activity and merge them in stable areas, thereby reducing the overall number of particles required in the simulation. Zhu and Bridson [78] use a hybrid approach called Particle in Cell (PIC)



(a) Eulerian *(b) Lagrangian*

Figure 2.1: Eulerian and Lagrangian apporaches. In the Eulerian approach fluid motion is represented as nett flow between cells, whereas in the Lagrangian approach the motion is produced by the particles themselves. In these diagrams the external force of gravity acts to pull the fluid downwards. Since there is more fluid on the left the buildup of higher pressure causes it to flow towards the right where the pressure is lower.

which advects particles according to an underlying grid. This is referred to as semi-Lagrangian since it incorporates both a grid and particles. Losasso [38] takes a different approach by modelling dense regions using grid-based level sets and sparse ones using particles.

An important property of fluids is that they are incompressible, and a realistic simulation needs to prevent compression artefacts. A step common to all fluid simulations is the computation of a pressure field, and this can be done in one of two ways: either by computing a Pressure Poisson Equation (PPE) or using an Equation of State (EOS). The PPE approach requires iteratively solving a system of linear equations ultimately producing a velocity field for the fluid, and this is generally the most time-consuming part of the simulation (Stam discusses this in more detail [64]). On the other hand EOS approaches compute pressure locally on each particle with a suitable stiffness value. EOS approaches are generally used in Lagrangian formulations, but PPE approaches can be used with either Lagrangian or Eulerian fluid models. Where they differ is in the targeted property of the resulting velocity field. A divergence free velocity field will prevent oscillation issues but might suffer from mass drift, while a density corrected field will preserve mass but allow oscillations at the free surface. EOS approaches can correct density and compression artefacts using Predictor Corrector schemes such as PCISPH presented by Solenthaler and Pajarola [62]. Ihmsen notes that solving PPE equations is generally impractical for computer graphics applications due to their prohibitive computational costs, but presents a novel scheme for significantly improving performance in [30].

The most costly step for Lagrangian simulation is usually calculating the neighbours of each particle, various schemes exist to accelerate this phase by storing particles in specialised data structures, we will cover these is more detail in section 6.4.

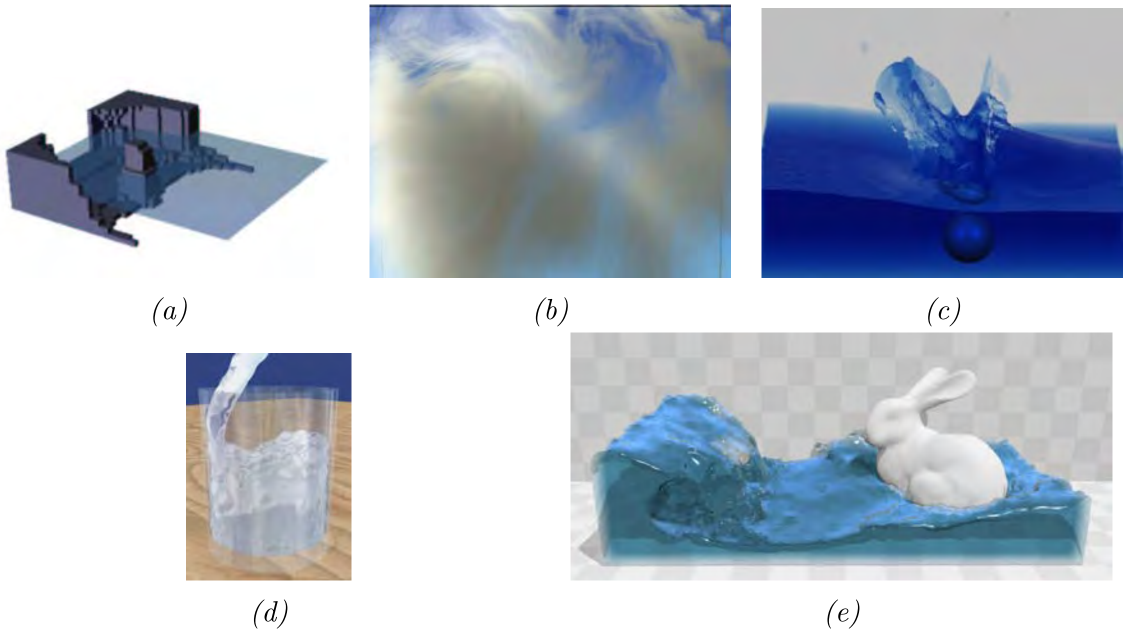
Parallelisation is the primary method of increasing performance of a simulation, but there are differences in how this can be applied to the two models. Because the cells in an Eulerian grid remain static relative to each other, the same cells will always interact with each other. This means it is possible to partition the simulation space up between nodes in a cluster, for example if the grid were 100x100x100 cells then we could slice it into 8 subcubes of 50x50x50 cells and simulate each subcube independently, only needing to share information about flow which occurs across the boundaries between out slices. This type of architecture would be perfectly suited to a cluster computing environment where nodes are connected via a network and communication time is high relative to computation time. Each node has separate memory and has to compute its flow and transfer results to its neighbours for each timestep, but since the slices are fixed each node knows which other node in the cluster handles the neighbouring slice and can send the relevant information directly. This is not the case for particle based Lagrangian simulations because at any point it is not possible to pre-determine which particles will fall into what regions of space. Thus a Lagrangian simulation is best suited to a shared memory architecture like a GPU where each core can access all the particles.

A final difference worth mentioning is that Lagrangian systems generally handle multi-phase phenomena such as melting and freezing more easily than Eulerian ones as shown by Muller et al [48].

2.2 Fluid Animation

The overarching concerns of fluid simulation for animation are stability and visual plausibility, and some compromise in full physical accuracy is often made to achieve these goals. Foster and Metaxas [22] present an early work featuring a grid-based 3D solution of the Navier Stokes equations. Their simulation was run offline on specialized workstation hardware, namely the Silicon Graphics Crimson R4000 which had a 100MHz processor and 256mb of memory [7]. It produced 20,000 frames of animation in 2.5 hours (2.2 frames per second) for a grid of 50x15x40. Marker particles are incorporated to track the surface and rendering is performed offline using RenderMan [6], a commercial rendering tool. Floating rigid bodies are incorporated after the simulation is complete by applying the pre-computed fluid forces. This has the benefit that different sizes and shapes could be tried for the objects without having to recompute the fluid motion, but the drawback is that the floating bodies cannot exert force back onto the fluid for full realism.

Stam [64] improves upon the method of Foster and Metaxas by replacing the explicit finite-differencing scheme with an unconditionally stable implicit solver. This is important because it allows significantly larger timesteps and allowing animators to freely adjust parameters without having the simulation “blow-up”. The method only deals with gaseous phenomena and does not track the surface of liquids. It also suffers from a non-physical amount of numerical dissipation, but this can be overcome by adding addi-



(a) (b) (c) (d) (e)

Figure 2.2: a) Foster and Metaxas [22] originally demonstrate the effectiveness of simulation for animation. They simulate a 3D scene of 50x15x40 with a fluid surface at 2.2 frames per second on a 100Mhz processor. Rendering is done in a separate pass and the time is not specified. b) Stam [64] achieves simulation and rendering at interactive framerates for grid sizes up to 30^3*, but the rendering does not track a free surface. c) Foster and Fedkiw [21] focus on extracting a high quality surface that is smooth in areas of low activity but also preserves detail in splashy areas. Rendering is performed separately using raytracing, while simulation takes several minutes per frame. d) Muller et al [41] simulates renders an opaque free surface at 20fps with 3000 particles. e) Macklin and Muller [39] present a fully realistic looking fluid simulation with 128k particles on a GTX680 GPU.*

tional external forces. Stam's system performs simulation and rendering grid sizes up to 30^3, and achieves framerates high enough for realtime user interaction. User interaction is why the stable solver is crucial. This system also used specialised hardware: an SGI Octane Workstation with 192mb of memory.

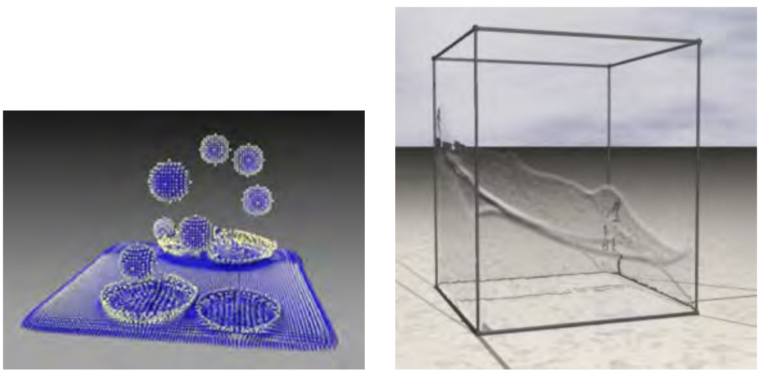
While Foster and Metaxas make use of particles to track the surface, these can suffer from visual artefacts as particles "pop" in and out of the surface. Foster and Fedkiw [21] address this issue using a hybrid particle and level-set method that ensures smooth surfaces in flat areas but maintains splashy detail.

Smoothed Particle Hydrodynamics is a popular Lagrangian technique originally developed by Gingold and Monaghan [24] for simulating galaxy formations and was introduced to the graphics community by Desbrun and Cani [20] to simulate deformable solids. Muller et al [41] present an interactive fluid consisting of 3000 particles running at 20fps on a 1.8 GHz Pentium IV with a Geforce 4 graphics card. Becker and Teschner [13] suggest an alternative density calculation method that reduces the compressibility of the fluid but imposes sever timestep restrictions and requires careful parameter tuning. Solenthaler and Pajarola [62] present an iterative, predictive-corrective scheme to eliminate compression artefacts between frames, thereby allowing larger timesteps. Ihmsen takes a different approach in [30] by discretizing Pressure Poisson Equations that exceeds the performance of PCISPH and allows very large scale simulations of up to 40 million particles.

2.3 GPU Accelerated Fluids

Harada et al [28] present the first full GPU implementation of SPH, though they cite previous works which performed neighbour search on the CPU. They achieve simulations of 60,000 particles running at 17fps and rendered using OpenGL point sprites, see figure 2.3a. They report that this is a 17x speedup compared to the equivalent CPU implementation, but the speedup increases along with the scale of the simulation for a maximum of 28x speedup with 262,144 particle. They also note that the maximum number of particles they can fit in their GPU's memory (768mb) is 4 million particles. This demonstrated the benefit of performing SPH simulation on the GPU but was cumbersome to implement, using shaders and encoding simulation data, such as pressure and velocity, into RGBA channels of textures. Goswami et al [25] present an SPH implementation using CUDA. They make use of Z-indexing to accelerate neighbour search and achieve comparable results of 15fps with 75,200 particles but this includes rendering of the free surface using raycasting. Zhang et al [76] present a multi-gpu system that provides roughly 3x speedup over a single GPU using 4 GPUs.

A particularly impressive result is shown by Muller and Macklin [39] which is a position based Lagrangian method that offers unconditional stability and improved performance over SPH by allowing arbitrarily large timesteps. They report simulations of 100k particles running in real time. Rather than calculating motion from forces, this method updates particle positions directly thereby avoiding instabilities and allowing more direct particle control. Since force and momentum are omitted some physical accuracy is sacrificed but the resulting motion is suitable for animation purposes.



(a) Harada [28] *(b) Goswami [25]*

Figure 2.3: Results from various GPU based fluid simulations.

2.4 Solid-Fluid Interaction

A fluid that cannot interact with other objects in its environment is severely limited in its usefulness for animation. Solid-fluid interaction is a broad area of research and there are various aspects to consider. There are various approaches that can be taken, each of which is able to model different effects with different levels of physical accuracy. Before reviewing existing work in this area we will cover some background material. Note that our discussion will be limited to implementations using the Lagrangian framework, and we refer the reader to Batty et al [12] for an explanation of Eulerian implementations.

There are two types of forces which can act between a solid and a fluid: hydrodynamic and viscous. Hydrodynamic forces exert pressure on the surface of an object and due to Newton's third law are exerted by an object back upon the fluid. These forces are the ones which give a floating object buoyancy or cause a sinking object to displace water surrounding it. Similarly these forces can push or move objects around. Viscous forces are caused by friction between the surface and the fluid, also known as drag. These forces can either slow a moving object or pull it along in a current (think of a boat hull designed to minimise drag).

A solid object is generally defined by a surface which the fluid must not penetrate, this surface can be represented in a number of ways (which will be covered shortly) and the object can either be static, moved kinematically, or moved by simulation. Static objects are obviously the simplest case since all that is required is a definition of the surface around which fluid must flow. Kinematic objects are those that move either under direct user control or along an animated path, in which case the surface position simply needs to be updated each frame. The most difficult case is objects which move along with the fluid, this is referred to as two-way coupling since the object exerts force to displace the fluid and the fluid exerts force to displace the object. The general approach is to move the fluid one simulation step, accumulate the forces applied to the object, move the object one simulation step and then repeat the cycle. We will confine ourselves to considering only non-deformable, non-breakable rigid bodies although discussion of elastic materials like cloth or rubber has been demonstrated by Akinci et al [10], and rupturing blood vessels are simulated by Muller [47]. We also refer the reader to Baraff and Witkin [73] for a more thorough explanation of simulating rigid body dynamics.

We will be using the Bullet Physics engine [2] to provide the motion and collision handling of the rigid bodies in our simulation. The major outstanding challenge is thus the representation of the surface shape of these objects. The simplest use case would simply be constraining a fluid to a container like a box, this can be done by simply reflecting particles back as they cross the planes defining the shape. While this is effective it has two problems: it introduces non-physical forces into the simulation which can result in undesirable "boiling" artefacts, and it is difficult to represent complex geometry this way. Kelager [34] uses and interesting approach of building tetrahedral meshes to represent shapes. Particles penetrating the mesh can be detected using barycentric coordinate tests against the tetrahedra, and the penetration can be resolved by applying forces to eject them out the appropriate face. While this approach works fine for analytic shapes like boxes or capsules, it is very difficult to construct an appropriate tetrahedral mesh for arbitrary shapes. An easier option is required. Since our fluid is made up of particles, the simplest approach is to construct the solids out of particles too, which are commonly referred to as boundary particles. There are a number of ways this can be done, but they all require a point cloud representation of solid particle positions. This can either be done by loading point positions directly, or by loading a triangle mesh representation and dynamically sampling the triangles on the fly. The benefit of dynamically generating the particles is that less storage is required for parts of solids that are not interacting with fluid. Also, a fundamental assumption of Lagrangian fluid simulations is that particles will be fully surrounded with other particles in order to calculate their forces. This is not true for particles on the free surface of the fluid, which means surface tension will not be accurately represented. One approach to rectifying this is to surround the surface of the fluid with "ghost air" particles thus providing for accurate density summations. Schecter and Bridson [61] use ghost air particles to simulate surface tension, showing a fluid that demonstrate adhesive and cohesive forces as it flows along the surface of a solid object whereas the particles would otherwise have dispersed in a spray.

Given a point cloud of boundary particles, the final issue is to calculate the actual forces applied from the fluid onto the solid and vice versa. Fluid particles exert pressure and viscous forces and are free to move relative to each other, but boundary particles must remain fixed relative to the object they are attached to. Thus forces applied to a solid surface depend on the orientation of that surface at a particular point. These forces can be broken down into normal and tangential components and a popular option for calculating them is the Monaghan Boundary Force (MBF) [43]. The normal component of the force corresponds to hydrodynamic forces, whilst the tangential component corresponds to viscosity. One drawback of the MBF is that it requires the normal to be stored for every boundary particle, although Akinci et al [11] propose an alternative boundary force calculation which avoids this requirement.

Thus, we now have the basic requirements for solid-fluid coupling in a Lagrangian model, but there are two final issues which need to be considered: Contact handling between multiple bodies, and guaranteeing non-penetration. Handling multiple bodies is difficult using a particle-based model because situations like stacked objects would be unstable, and collisions between objects would not be physically accurate. Oh et al [54] address this issue with an impulse-based boundary force between their solid particles, but in our implementation we will simply wrap all solid objects with Bullet collision shapes allowing the Bullet engine to handle contact solving.

The second issue of guaranteed non-penetration is important to avoid unsightly artefacts and leaking. As always there is a tradeoff between timestep size and fidelity. If solid particles are sufficiently close then penetration can be mostly avoided with small timesteps, but the simulation will run very slowly. Also, when penetration does occur sitff penalty forces are used to push particles back out, which results in excess energy being introduced. A solution used by Becker et al [14] is to use predictor-corrector system between timesteps to calculate the future position of particles and correct penetration artefacts before they occur. Their system robustly handles fast-moving solid objects such as a stone skipping off the surface of water.

Summary

- There are two basic models of fluid simulation, Eulerian is grid based and Lagrangian is particle based.
- Eulerian methods are good for taking physically accurate measurements at points in space and they parallelise very well on non-shared memory architectures like clusters.
- One drawback of Eulerian methods are that they require a linear equation solver to preserve mass which is tricky to implement. Also high resolution grids are needed to capture fine detail, which might waste memory if the fluid only occupies a small proportion of the domain.
- Lagrangian methods are easier to implement and more flexible, being able to trivially preserve mass and easily interact with solid objects. The major bottleneck is neighbour searching which is an important area of research using acceleration grids.
- In practice the two methods are often combined.

Chapter 3

GPU Computing

3.1 Introduction

Graphical Processing Units (GPUs) are add-in cards that accelerate the computationally intensive task of rendering complex 3D graphics, which involves processing large of amounts data in a massively parallel fashion. Early generations of hardware were hard-coded for specific graphics related tasks, but over time graphical applications began to push the boundaries of these fixed functions. As demand increased for more flexibility, the GPUs gradually became fully programmable. This means that apart from allowing more flexible rendering options, the computing power of GPUs can also be applied to other types of high performance computing problems. This is referred to as General Purpose GPU or GPGPU computing. Examples can be found in bioinformatics, computational chemistry, financial modelling and various other fields. Initially such computing could only be achieved by expressing problems in in terms of graphical APIs such as OpenGL and DirectX (these will be discussed in more detail in Chapter 5) but, as the adoption of GPGPU became more widespread, tools were developed to allow general problems to be expressed more naturally.

A popular example of such a a toolkit is CUDA, developed by the NVIDIA Corporation [51], which provides a compiler for a C-like language called CUDA C and allows the GPU to be used as a co-processor to which the CPU offloads computationally expensive tasks. Since the GPU is a separate device, the basic idea is to copy data onto it, execute a program which launches many threads to process the data, and then copy the results back. These programs are called CUDA kernels. In this chapter we will discuss the overall evolution of GPU hardware as well as describing how CUDA programs are launched and executed, which is referred to as the CUDA Programming Model. Where relevant we will highlight points that are applicable to our current application of fluid simulation, and finally we will mention some points to consider for achieving optimal performance. The full details of our simulation implementation can be found in Chapter 6.

3.2 GPU Hardware Description

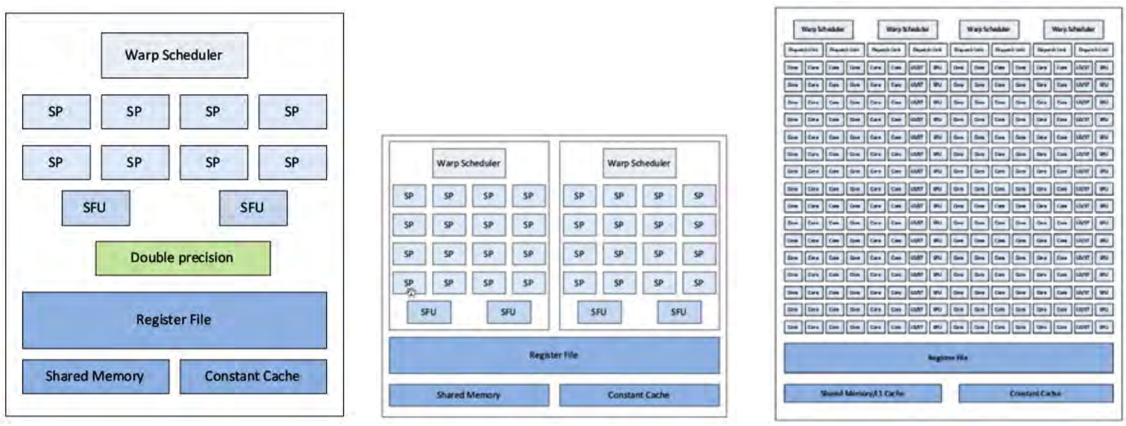
Because of the nature of the task it needs to solve, a GPU is designed very differently to a CPU. In broad terms, a CPU generally executes many different concurrent tasks and has to switch rapidly between multiple different processes. On the other hand a GPU processes large batches of homogeneous graphics data usually applying the same operation to each piece of data. Because of these differences in tasks, there are a number of fundamental design trade-offs which we will illustrate by comparing the specifications of a top-of-the-range example of each, namely the NVIDIA Geforce GTX 960 [52] and the Intel Core i7 [33]. Although there are specialised professional products available in both areas, we will focus exclusively on consumer-grade hardware. A typical desktop CPU will have a handful of cores (usually 4 to 8) but a GPU will have many times more, in our case 1024. There are however significant differences between the processing cores of a CPU and a GPU.

Modern CPUs use sophisticated optimization tricks such as pipelining, branch prediction, speculative execution and out-of-order execution to increase the performance of linear, single threaded applications. Implementing these techniques requires a lot of extra complex circuitry, which takes up room on a chip and limits the number of cores that can fit. The reason many more cores can fit onto a GPU is that they are much simpler and do not implement these types of optimizations. Another very important difference is the amount of space dedicated to cache memory. A large cache means that instructions which are executed or data which is operated upon frequently can be stored on the chip, this avoids them being repetitively fetched from memory which would waste many cycles waiting for values to arrive, from 200 to 800 cycles depending on generation [50]. We will examine later how caching is a significant factor in GPU performance. Finally, the clock speed of GPU chips tends to be much lower, around 1Ghz versus 3Ghz for a CPU. Nevertheless, the GPU can achieve significantly higher floating point operations per second (FLOPS) thanks to its larger number of cores and higher memory bandwidth.

While not essential to developing CUDA applications, an understanding of how these cores operate helps in achieving maximum performance. We will briefly describe the evolution of the NVIDIA range of hardware, full details of which can be found in the CUDA Handbook by Wilt [72]. Each core is called a *Streaming Processor* (SP) and they are arranged in groups called *Streaming Multiprocessors* (SMs). The layout of the SMs has changed over time as NVIDIA has refined the architecture, and at the time of writing there are three major generations namely Tesla (1.x), Fermi (2.x) and Kepler (3.x). These are depicted in figure 3.2. Notice that the third generation (Kepler) is vastly different to the other two and is therefore labelled SMX to highlight this. The elements that make up each generation are basically the same and what differs is the number of them, each SM will generally be made up of a set of SPs, a warp scheduler and a register file. The registers are similar to registers on a CPU but there are many more of them available. The term *warp* originates from fabric weaving and denotes a group of threads, it is the smallest unit of execution in the CUDA environment. At the time of writing all CUDA warps are of size 32, meaning that regardless of how many threads have been launched by the kernel only 32 are actually executing at any point in time per SM. The implications of this will be discussed in more detail in section 3.3 but the

Figure 3.1: A schematic comparison of the layout of a CPU chip to a GPU. The CPU has only a few cores but a lot of space dedicated to the control unit and a single large cache. The GPU on the other hand has several smaller and simpler control units, each shared by many cores. The GPU also has much smaller caches associated with each control unit. Diagram from Cuda Programming Guide [50]

important thing to understand is that all the SPs within an SM share the same scheduler, which means that they all execute the same instructions at once. This is referred to as a SIMT architecture for Single Instruction Multiple Thread and is analogous to SIMD on the CPU. Sharing rather than duplicating the scheduling circuitry allows more cores to be included, but it does have certain performance implications, especially when dealing with branching code as we shall see in section 3.4.



(a) *SM 1.x (Tesla)* (b) *SM 2.x (Fermi)* (c) *SMX 3.x (Kepler)*

Figure 3.2: The layout of Streaming Multiprocessors in different generations of NVIDIA GPU hardware. Diagram from Cuda Handbook [72]

Apart from the scheduler and registers, each SM also contains shared blocks of cache memory as well as a Special Function Unit (SFU) to accelerate expensive maths functions like trigonometry and square roots. The full details and specifications of the SMs can be found in the CUDA Programming Guide [50]. We will now see how the CUDA programming model maps onto this underlying architecture.

3.3 CUDA Programming Model

In CUDA the GPU is referred to as the *device*, while the CPU is referred to as the *host*. Code destined to be executed on the device is written in CUDA C, an extension of the C programming language that allows the programmer to define special functions, called *kernels*, to be executed on the GPU. Executing a kernel launches a group of threads on the GPU, each running a separate instance of the code. Each thread also defines a built-in integer identifier, known as a thread ID, which can be used for indexing into arrays of data. This way the same operations can be applied simultaneously to all elements. As an example we may have an array of one thousand elements and we wish to multiply each element by 2. In a serial program we would loop over each element in turn, but using CUDA we launch a thousand threads with IDs from 0 to 999, each of which corresponds to the index of a single element within the array. The syntactic details of how to launch kernels are covered in the CUDA Programming Guide [50].

While we can conceptually think of all threads executing at once, in reality the device groups the threads into *blocks* and executes a certain number of these at once. How many depends on the hardware on which the code is being executed: a more powerful device like a workstation GPU will execute many blocks at once, whereas a smaller device like a laptop of cellphone will only execute a few. A single kernel launch will contain a number of blocks, which is specified by the programmer. All of these blocks together make up a *grid* and within the grid each block has its own *block ID*. Using this strategy a large dataset could be subdivided and each element referred to using a unique combination of block ID and thread ID. To solidify this concept let us consider an SPH simulation containing 5000 particles. For each particle we would need to store values such as position and velocity, this can be done by simply storing a one dimenional array of 3 element vectors. To process all the particles a program could launch a grid of 5 blocks each containing 1000 threads. Otherwise it could launch 50 blocks each with 100 threads, or any combination in between. The question of what size to make the blocks is determined by the concept of *occupancy*. Before discussing this let us quickly note that the maximum number of thread per block is 65535 and the maximum number of blocks per grid is 1024. Blocks and grids can also be multi-dimensional to fit variously shaped data, figure 3.3 shows a two dimensional grid of two dimensional blocks.

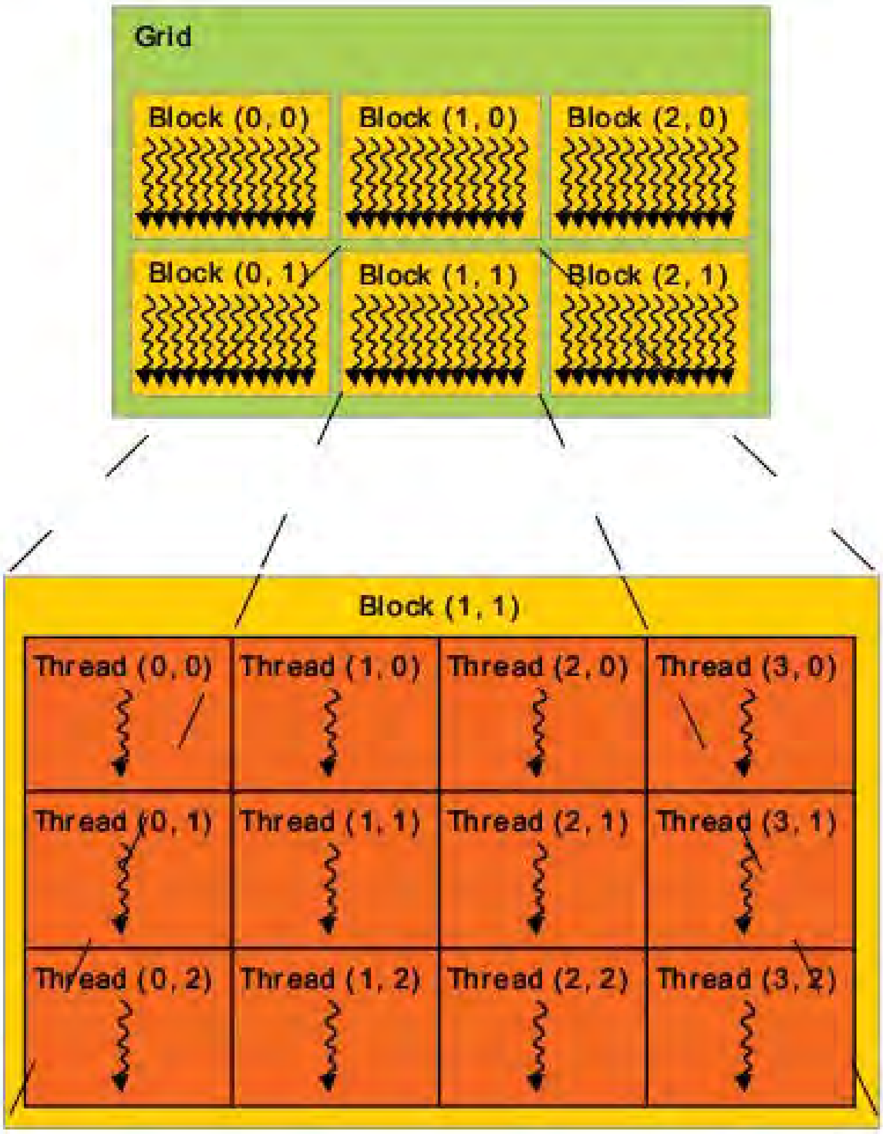


Figure 3.3: Diagram from the CUDA Programming Guide [72] showing a two dimensional CUDA grid made up of 2 rows of 3 columns of blocks. Each block contains 3 rows of 4 columns of threads.

3.3.1 Occupancy and Latency Hiding

At any point in time, a particular SM will be actively executing a single warp of 32 threads, but as a program proceeds it may reach a point where it needs to fetch values from memory. This can take many hundreds of clock cycles, during which execution may stall if there is nothing else to do, thus causing latency. In order to make maximum use of the hardware there must always be some threads which are ready to execute while others are waiting. This is called *latency hiding* and is achieved by always having multiple blocks ready to execute. From the programmers perspective the blocks must be considered completely independent from each other and able to execute in any order, which allows them to be swapped out with each other. This mechanism serves the same purpose as instruction reordering or speculative execution on a CPU, ensuring that useful work continues while values arrive from memory. In order for the scheme to be most effective, there need to be as many blocks as possible to switch between. This is referred to as *occupancy* and is one of the main keys to achieving maximum performance. Occupancy is defined as the ratio of the number resident warps to the maximum number of resident warps (the maximum values are given in the programming guide). Occupancy is limited by two factors: register usage and shared memory usage.

Part of achieving good latency hiding is ensuring that context switching between threads happens as quickly as possible. In a CPU task switching generally requires saving the execution context of one thread of execution to RAM and then loading another, both operations involving reading and writing all the registers which is time consuming. Due to the large cache and optimization tricks mentioned above, a CPU can expect to follow a thread of execution for many cycles before stalling. This amortizes the cost of task switching. A GPU on the other hand has thousands of times more threads and thus cannot afford to load and store all of their execution states repetitively. Thus the GPU simply keeps the execution state of all threads within a block resident throughout its execution. This makes context switching essentially cost-free.

While there are a large number of registers available within each SM, they remain finite and each thread in a block takes up a certain amount of them depending on how many variables it has. This in turn increases the size of the blocks themselves and the more registers a block uses the fewer blocks can be resident in each SM, meaning latency hiding will be less effective. In terms of an SPH simulation this means that careful consideration needs to be given to what attributes of each particle are stored, for example the movement of a particle can be calculated using the position and density of surrounding particles, this requires a floating point vector and a floating point scalar to be stored per particle as registers in each thread of a kernel. If the temperature were to be taken into consideration then an extra floating point register would be required which would decrease the number of blocks that could be resident at once. The exact number of resident blocks according to how many registers they contain can be worked out using the CUDA Occupancy Calculator, which is a spreadsheet that can be downloaded from the CUDA website [51].

In principle having many smaller blocks should increase occupancy and therefore performance, although this is not necessarily always the case and careful benchmarking is required to optimize real-world scenarios [70]. If a grid is launched whose dimensions require more registers than are available then a phenomenon known as *register-spilling* occurs. This means excess variables that do not fit in the register files are offloaded to RAM, which is much slower. This can have disastrous performance implications, but it does mean that at least the kernel can continue executing. This makes the register file a soft-limit to occupancy, whereas a hard limit is imposed by the shared memory. This is a portion of cache that is allocated per block and will be covered below.

3.3.2 Memory Hierarchy

Within the CUDA programming environment there are several different types of memory available each with different sizes and performance characteristics. As noted above the registers reside on the chip within the SM and are the fastest memory but most limited in size. For example the Kepler architecture has 64K registers and supports a maximum of 2048 threads, which means that at full occupancy each thread can use 31 registers (64000/2048=31.25). Each register is 32 bits which corresponds to a single precision floating point value. If a kernel defines more variables than this then either register spilling will occur, or the number of threads per block needs to be reduced.

Due to their limited size, registers are not appropriate for storing large amounts of data, for this purpose there is global memory which resides off chip in RAM modules on the graphics card. Global memory is much larger, up to 2G in most devices and is used to store large datasets on which the CUDA kernels will operate. It is allocated and managed similar to heap memory in C using the functions *cudaMalloc()* and *cudaFree()*. Access to global memory incurs two types of overhead: firstly the inherent latency of fetching values from physically separate RAM, but secondly potential overfetch due to memory alignment requirements. Global memory is fetched in transactions of either 32, 64 or 128 bytes. This means that fetching values smaller that the alignment size can introduce transfer overhead as extra bytes surrounding the desired value will also be fetched, and also fetching values from non-contiguous addresses will involve transfer overhead for bytes fetched from every separate location. This transfer overhead is minimized by ensuring that all desired values are as contiguous as possible, in which case the memory accesses will be said to *coalesce*. We will come back to the topic of coalescence in section 3.4). Our SPH simulation, described in chapter 6 makes use of global memory to store fluid particle data. A final point worth noting is that the memory on the GPU is fixed in size, i.e. there is no paging like on the host to give the illusion of unlimited memory space. This is a relevant consideration in very large scale simulations (millions of particles), but for our specific purposes the number of particles is significantly limited by the goal of maintaining realtime performance.

In order to alleviate the latency introduced by global memory fetches, there are several layers of cache memory available. Firstly there is the constant cache, shared by the whole SM, which stores values marked by the compiler as unchanging. Next there is an L1 cache which resides on each SM. This cache usually operates transparently to the programmer although part of it can also be allocated for manual control, referred to as *shared memory*. Shared memory is visible to all threads within a block and it is possible to use it to cooperatively share data amongst threads to increase performance. The idea is that if multiple threads are going to operate on the same memory value then, rather than fetching it repetitively for each thread. We will describe an application of this in more detail in chapter 6. In the latest architecture, Kepler, the size of the L1 cache and shared memory can be configured as either 16kb shared/48kb L1 or 16kb L1/48kb shared or 32kb each. Finally there is an L2 cache of 1.5mb in Kepler which is shared among all SMs.

Apart from the various levels of caching there is also *texture memory*. One of the most frequent operations performed by graphics cards is shading, which means applying 2D textures on to surfaces. This usually involves sampling colour values from regions that are close together, for example if a group of threads is shading the top left corner of a square then they will all be accessing the corresponding part of the texture being applied. In most programming languages the method for storing a 2D texture would be in row-major array of colour values. Unfortunately this means that addresses which are close together in coordinate space might be far apart in address space. Since graphics cards deal primarily with texturing 2D surfaces they contain dedicated hardware for accelerating the process. The most important thing to note about texture memory is that it is stored in a way that optimizes spatial locality.

In chapter 7 we make use of the texture memory for both of our rendering implementations, our splatting solution uses multiple passes which store intermediate results in 2D texures, while our raymarching implementation makes use of 3D texture memory to store the fluid density field being rendered. An important point to note is that our simulation makes use of both the CUDA and OpenGL APIs each of which control their own memory and data structures on the GPU. This means that interoperation needs to be carefully managed, this is described in section 7.1.2.

3.4 Performance Considerations

Although the CUDA programming environment make correct code simpler to implement on the GPU, some care still needs to be taken in order to achieve maximum performance. The CUDA C Best Practices Guide lists a number of concepts which need to be considered, and we will outline some of the more important ones, namely coalescing and divergence. Recall that access to global memory occurs in fixed size transactions. If the thread within a warp all access contiguous addresses in memory then this results in fewer load transactions being required since several values can be loaded at once in a single transaction. On the other hand if each warp were to fetch values from different regions of memory then overhead is incurred by fetching surrounding unused values. As an example consider a kernel which fetches a 4 byte float value for each kernel, if each thread's value is separate then 28 bytes of unused data is fetched for each value reducing memory throughput by 8 which will have a significant performance impact. With this in mind it is also beneficial to consider the layout of data in memory, if data is made up of elements containing several attributes a natural way to store them might be using structs. If however a kernel operates on only one value at a time then fetching the entire struct might involve transferring unused values. Therefore rather than storing data in an array of structures, it is usually better to store it in a structure of arrays. This way only the attributes relevant to the currently executing kernel are fetched from memory.

A second important consideration is minimizing divergence. Since all SPs within SM execute the same instructions, handling of branching code can be costly. If some of the threads within a warp follow one branch and the rest follow another then the instructions for them will need to be executed serially thereby reducing performance. The way to prevent this is to ensure that the data on which each block operates is layed out in such a way that contiguous pieces of data will follow the same branch.

Chapter 4

Smoothed Particle Hydrodynamics Theory

4.1 Description of Navier Stokes Equations

The Navier Stokes equations describe the motion of a fluid over time and thus form the basis of any fluid simulation technique. Specifically they model a small volumetric fluid element referred to as the *material derivative* that moves with the flow of the rest of the fluid. Depending on whether the Eulerian or the Lagrangian model is used, these equations may have different interpretations. In the Eulerian viewpoint the fluid is described in terms of a three dimensional pressure field through which it flows. Pressure is a scalar value and this field simply contains the pressure values at a set of fixed points in space. These pressure values evolve over time and influence each other to produce fluid motion. In the Lagrangian model the fluid is represented as discrete particles which interact with each other while moving through space. Ihmsen provides more detail comparing the two approaches in the appendix of [31].

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{4.1a}$$

$$\rho(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}) = -\nabla P + \mu \nabla^2 \mathbf{v} + \rho \mathbf{g} \tag{4.1b}$$

Equation (4.1a) is the *mass* conservation equation ensures that the total mass of liquid in a system does not change over time. In the Eulerian framework this is enforced by ensuring that at every time step the amount of fluid flowing into a cell equals the amount flowing out. This is also referred to as “zero-divergence” and is required to simulate an incompressible liquid.

Since we are working in a particle-based Lagrangian framework the mass of the fluid is represented by the mass of the particles. Since there are a fixed number of particles within a simulation the total mass in the system will always be conserved and thus equation (4.1a) is satisfied.

Equation (4.1b) is the *momentum* conservation equation. It describes how fluid flows under the influence of three types of force: *pressure*, *viscous* and *external*[1]. Each force is described by one of the three terms on the right hand side. We will refer to these forces as $\mathbf{F}$ with appropriate superscripts:

$$\underbrace{-\nabla P}_{\mathbf{F}^{\text{pressure}}} + \underbrace{\mu\nabla^2\mathbf{v}}_{\mathbf{F}^{\text{viscosity}}} + \underbrace{\rho\mathbf{g}}_{\mathbf{F}^{\text{external}}} \tag{4.2}$$

As mentioned in the notation guide the first and second term refer to the gradient of the pressure and the Laplacian of the velocity respectively. We will see how each of the force terms is calculated in more detail in section 4.3, but first we will need to define the concept of SPH interpolation in section 4.2.1 which is used to sample pressure and viscosity values at points in space.

For now let us examine the left hand side of equation 4.1b. The terms inside the brackets together form the *material derivative.* [2] This describes the rate of change of a quantity over time. In this case it refers to the change in velocity of a fluid particle. In particle-based solvers this term can be replaced by the shorter form as follows:

$$(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}) \rightarrow \frac{D\mathbf{v}}{Dt}$$

Since this work concerns SPH which is a particle-based approach, we will substitute this into (4.1b) thus giving:

$$\rho\frac{D\mathbf{v}}{Dt} = -\nabla P + \mu\nabla^2\mathbf{v} + \rho\mathbf{g} \tag{4.3}$$

Finally we can move the density factor ρ across to the right hand side. The physical interpretation of this is that a denser fluid will dampen the other forces.

$$\boxed{\frac{D\mathbf{v}}{Dt} = \frac{-\nabla P}{\rho} + \frac{\mu\nabla^2\mathbf{v}}{\rho} + \mathbf{g}} \tag{4.4}$$

This gives us the final formulation which we shall use as the basis for our simulation. In this form is effectively a re-arrangement of Newton's Second Law (from $\mathbf{F} = m\mathbf{a}$ to $\mathbf{a} = \frac{\mathbf{F}}{m}$). In section 6.3 we will see how and can be to form the basis for advancing particle positions.

[1] This term mainly refers to gravity, hence $\mathbf{g}$, but for implementation purposes it also includes boundary forces such as walls and obstacles.

[2] Interestingly, it is the non-linear $\mathbf{v} \cdot \nabla \mathbf{v}$ (known as the convective term) which is the main source of difficulty in providing an analytic solution to these equations, making them the subject of one of the well known Millennium Prize Problems. We neatly sidestep this issue by not being remotely concerned with rigorous analysis, only straightforward practical application.

4.2 SPH Formalisation

4.2.1 Interpolating Continuous Fields from Discrete Points

In equation (4.2) we refer to the terms $\mathbf{F}^{\text{pressure}}$ and $\mathbf{F}^{\text{viscosity}}$, now it is time to explain how such values are derived. While a real world fluid has pressure and density throughout, for simulation purposes we can only store these values at discrete points in space. In order to measure the values in-between these points the SPH concept applies a smoothed interpolation kernel which essentially blurs the values. This is visualized in figure 4.1.

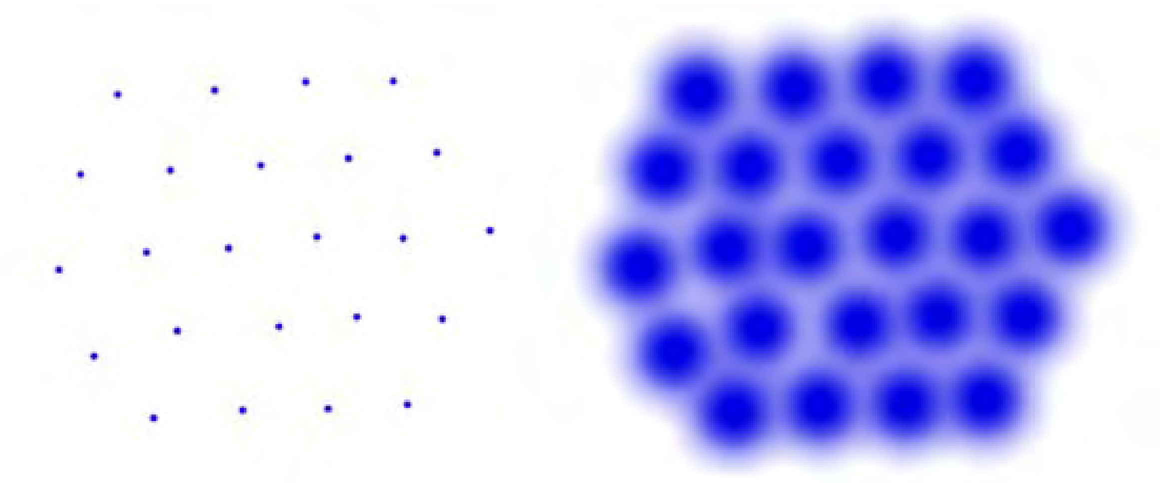


Figure 4.1: The particles on the left are defined only at discrete points in space. If measurements were made between these points then no values would be found. By interpolating the values, SPH provides a way of taking measurements at any point within the continuous field.

The key to obtaining such continuous values is the concept of integral interpolation. Given a set of point values, let's say that we wish to calculate the value of a function A at some arbitrary point $\mathbf{r}$ in space. The integral interpolant version of this (A_I) would be:

$$A_I(\mathbf{r}) = \int A(\mathbf{r}')\, W(\mathbf{r} - \mathbf{r}', h)\, \mathrm{d}\mathbf{r}'$$

where $\mathbf{r}'$ are the values for the set of points we have. In section 4.3 we will replace the abstract quantity A with pressure and viscosity. Since we are computing these values numerically we will have to emulate an integral with a discrete summation (A_S) which gives us:

$$A_S(\mathbf{r}) = \sum_j V_j A_j W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.5}$$

This is essentially the same as the integral form (A_I) except you will notice that we have introduced a new factor V_j. This represents volume around a point and roughly speaking it can be thought of as taking the place of the differential $\mathrm{d}r'$. We also note that volume can be expressed in terms of mass and density:

$$V = \frac{m}{\rho}$$

and so we can substitute this back into equation 4.5 giving the final form of the SPH interpolation equation::

$$\boxed{A_S(\mathbf{r}) = \sum_j \frac{m}{\rho} A_j W(\mathbf{r} - \mathbf{r_j}, h)} \tag{4.6}$$

This equation will be modified later to calculate the pressure and viscosity. The pressure is based on density which is in turn derived from particle mass. The first application of the SPH interpolation will be to calculate density at a position as follows:

$$\rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.7}$$

4.2.2 Finding Field Gradients Using Kernel Derivatives

An important thing to note is that in solving the Navier Stokes equations we refer to the gradient of the pressure field and the Laplacian (second derivative) of the viscosity field. So far we have seen that SPH interpolation can be used to find the value of the quantity field A, but Muller also shows in [45] that we can find the gradient and Laplacian of A simply by finding the gradient and Laplacian of the kernel. Thus we have:

$$\boxed{\nabla A_S(\mathbf{r}) = \sum_j \frac{m}{\rho} A_j \nabla W(\mathbf{r} - \mathbf{r_j}, h)} \tag{4.8}$$

and:

$$\boxed{\nabla^2 A_S(\mathbf{r}) = \sum_j \frac{m_j}{\rho_j} A_j \nabla^2 W(\mathbf{r} - \mathbf{r_j}, h)} \tag{4.9}$$

4.3 Breakdown of Navier Stokes Terms

Pressure Term

$$\mathbf{F}^{\text{pressure}} = -\nabla P \tag{4.10}$$

The pressure term describes the force exerted on a particle by the pressure of its surrounding particles, which in mathematical terms is the gradient of the scalar pressure field. Since water flows from regions of high pressure to regions of low pressure, flow will proceed along the negative gradient as if down a hill. First we will see how the pressure of a single particle is determined then we will see how the pressure gradient field surrounding that particle is found and thus the force exerted on it.

Ideal Gas Equation To calculate the pressure of a fluid particle we begin with the Ideal Gas Equation. This is an equation of state (EOS) which describes the properties of a fixed mass of gas.

$$PV = nRT$$

where n is the number of molecules, R is the universal gas constant and T is the temperature. Since all three of these values remain unchanged in a basic simulation we can replace the right hand side with a single constant k.

$$PV = k \tag{4.11}$$

For the purposes of simulation the term "fluid" can refer to either a liquid or a gas. In this thesis we are primarily concerned with liquids, which display two important properties which need to be considered: a liquid resists compression and it does not expand to fill an available volume. Together these two properties can be referred to as the "rest density" which means a given mass of liquid will maintain a fixed volume. Intuitively this can be pictured by imagining a large sealed bottle with a small amount of water. In a liquid state the water will pool at the bottom (fixed volume, rest density) but if heated to gaseous steam it will expand to fill the container (larger volume, lower density).

Rest Density SPH was initially developed for large-scale astrophysical simulations, where particles behave in a gas-like manner, thus it needs some modification to properly represent a liquid. In its original form the pressure equation 4.11 produces only positive values for P since ρ and k are always positive. This represents purely repulsive forces. In order to maintain a constant volume we need to introduce attractive forces in the form of negative P. With both repulsive and attractive forces the liquid will demonstrate internal cohesion rather than simply expanding. As proposed by Desbrun [19] we introduce a rest pressure P_0 into equation 4.11.

$$(P + P_0)V = k$$

Since a particle represents a fixed mass, we can express volume in terms of density by replacing V with $\frac{1}{\rho}$.

$$
\begin{aligned}
(P + P_0)\frac{1}{\rho} &= k \\
P + P_0 &= k\rho
\end{aligned}
$$

Now we can express the rest pressure in terms of rest density $P_0 = k\rho_0$.

$$
\begin{aligned}
P + k\rho_0 &= k\rho \\
P &= k(\rho - \rho_0)
\end{aligned}
\tag{4.12}
$$

Thus, is we substitute $-\nabla P$ for A in equation 4.8 we can work out the pressure force, which is the gradient of the pressure field, as follows:

$$
\begin{aligned}
\mathbf{F}^{\text{pressure}} &= -\nabla P(\mathbf{r}) \\
&= \sum_j m_j \frac{P_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r_j}, h)
\end{aligned}
\tag{4.13}
$$

This however would yield an unbalanced force. Consider if we have two neighbouring particles a and b. When a calculating its pressure it will only look at the pressure exerted by b and similarly b will only use the pressure exerted by a. We would expect that physically the two particles should exert the same force on each other, but, since the pressures at the two particles are different in general, this will not be the case. We can address this by averaging the pressure of the two particles as follows:

$$
\begin{aligned}
\mathbf{F}^{\text{pressure}} &= -\nabla P(\mathbf{r}) \\
&= \sum_j m_j \frac{P_i + P_j}{2\rho_j} \nabla W(\mathbf{r} - \mathbf{r_j}, h)
\end{aligned}
\tag{4.14}
$$

Viscosity Term

$$
\mathbf{F}^{\text{viscosity}} = \mu \nabla^2 \mathbf{v}
\tag{4.15}
$$

Viscosity is a frictional force, and intuitively it can be thought of as the difference in motion between two points of a fluid. Mathematically this is expressed as the second derivative of the velocity field, multiplied by scalar a weighting factor μ (also known as the dynamic viscosity). Similar to the pressure term we can thus substitute $\mu\nabla^2\mathbf{v}$ for A in equation 4.8 to get:

$$
\begin{aligned}
\mathbf{F}^{\text{viscosity}} &= \mu \nabla^2 \mathbf{v} \\
&= \mu \sum_j m_j \frac{\mathbf{v}_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r_j}, h)
\end{aligned}
\tag{4.16}
$$

This suffers from the same symmetry issue as the pressure term, which can be addressed in the same way by averaging the velocity between the two particles:

$$
\begin{aligned}
\mathbf{F}^{\text{viscosity}} &= \mu \nabla^2 \mathbf{v} \\
&= \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r_j}, h)
\end{aligned}
\tag{4.17}
$$

External Term

The external forces are generally constant and do not need to be recalculated at each step. Usually gravity is the biggest external force acting on a liquid, hence the term is g.

4.4 Smoothing Kernel

Whilst the density and viscosity parameters determine the major characteristics of the simulated material, the properties of the kernel function W ensure that the simulation remains numerically stable and particles do not behave erratically. The kernel takes $\mathbf{r}_{ij}$ as its input which is the offset vector between particles i and j. It also takes in h which is the cutoff radius, outside of which particles do not interact. By assuming that particles are only influenced by nearby particles (rather than all others) we can significantly reduce the algorithmic complexity of evaluating equation 4.5 from $O(n^2)$ to $O(nm)$ where m is the average number of particles within radius h.[3]

Desbrun [19] cites some works which use the Lennard-Jones potential for inter-particle forces, seen in figure 4.2a. This function models at a microscopic level the interaction between molecules of a material or atoms within a molecule. It was originally chosen for its simplicity and computational efficiency, but there were drawbacks: firstly that the parameters are unintuitive to manipulate and secondly that due to the focus on molecular or atomic level structure it was difficult to achieve any specifically desired macro level behaviours (for example: make the water appear more "splashy" for dramatic effect). Secondly the interaction force has an inflection point representing the rest distance between particles, in a numerical simulation this can cause unwanted oscillations or clumping.

[3] Chapter 6 describes how implementing this threshold can be accelerated using neighbour search grids

In order to produce more pleasing macroscopic behaviour, Gingold and Monaghan used a Gaussian kernel in their original formulation [42] which is given below and shown in one dimension in figure 4.2b. [4]

$$W(\mathbf{r}_{ij}, h) = \frac{1}{h\sqrt{\pi}} e^{-((\mathbf{r}_{ij})^2/h^2)}$$

Monaghan also introduces spline based kernels [42] as an alternative, which are more computationally efficient. Any kernel must have three important properties in common with the Gaussian function though:

1. It must be normalized, i.e. its integral must sum to one.
   This prevents the introduction of phantom forces.

$$\int W(\mathbf{r}', h)\, d\mathbf{r}' = 1$$

2. It must be positive over its entire domain.
   This means particles have no attractive forces, only repulsion.

$$W(\mathbf{r}', h) \geq 0$$

3. It must be even, i.e. symmetric around each axis.
   $\mathbf{r}'$ is a vector and we want it to behave the same in any orientation.

$$W(\mathbf{r}', h) = W(-\mathbf{r}', h)$$

As we saw in section 4.2.1 a kernel W is used in the summation interpolant equation 4.5 to calculate three field quantities: density, pressure and viscosity. These applications all have slightly different requirements and so we will use a separate kernel for each, bearing in mind that the kernels need to fulfil the criteria laid out above.

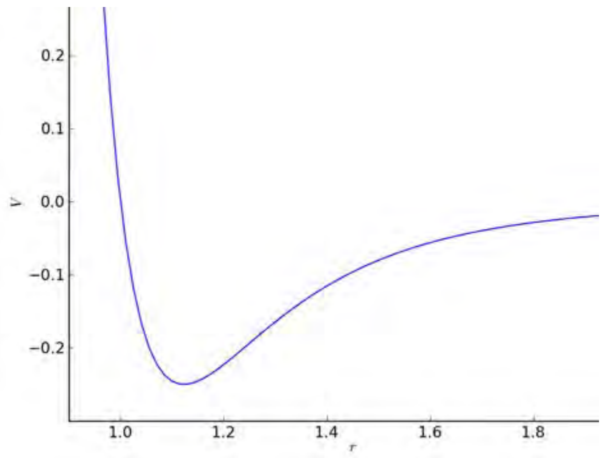
Below we list the formulas for each kernel and what it is used for. The derivative (∇W)) of the pressure kernel is given since we need the gradient of the pressure field to calculate $\mathbf{F}^{\text{pressure}}$ in equation 4.2. Similarly we use the Laplacian of the velocity field to calculate $\mathbf{F}^{\text{viscosity}}$ so we will give the second derivative ($\nabla^2 W$) of the viscosity kernel.

In figure 4.3 the kernels are plotted in two dimensions since it is easier to visualize. In practice the three dimensional versions can be implemented by simply applying the two dimensional equation one axis at a time (thus, in the equations listed below, the input values r and v are scalar).

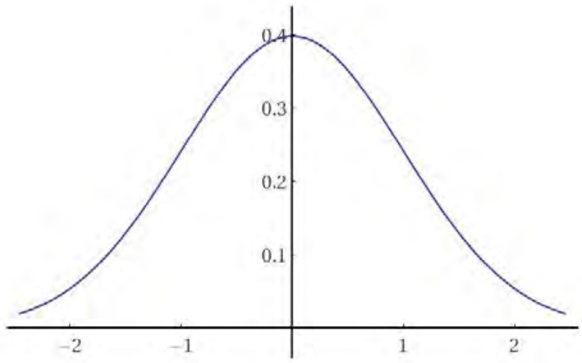
Density Kernel (aka Poly6) *Used to smooth density field.*

$$W_{\text{poly6}}(r, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |r|^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

[4] For implementation purposes it is valid to split the axes and deal with them individually, as we shall also discuss further in Chapter 6.



(a) Lennard Jones



(b) Gaussian Function

Figure 4.2: Two smoothing kernels originally used for SPH. The Lennard-Jones potential is used to model inter-molecular forces, but is unsuited for simulation because it has an inflection point which may cause clumping. The Gaussian function does not have this problem but is computationally expensive to evaluate. Specially designed kernels which solve both these issues are listed in figure 4.3

This kernel was designed by Müller et al. [45] with two improvements over the standard Gaussian function. Firstly it is cheaper to calculate since it does not contain any square roots and secondly it vanishes to zero at the boundary radius h (since $h^2 - |r|^2 = 0$ when $h = r$). This second point improves stability over the Gaussian function which only asymptotically approaches zero. This kernel is used to smooth the density field which will later be used to calculate the pressure field.

Pressure Kernel (aka Spiky) *Gradient* ∇ *used to smooth* $\mathbf{F}^{pressure}$

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

$$\nabla W_{\text{spiky}}(r,h) = -\frac{45}{\pi h^6}(h-|r|)^2 \tag{4.19}$$

The problem with the poly6 density kernel is that its derivative is zero at the center. In the terms of the Navier Stokes equation 4.2 we can see that $\mathbf{F}^{\text{pressure}}$ is calculated as the with the gradient of the pressure field. This means that if the poly6 kernel were used to smooth the pressure field then there would be a dead point where no pressure was applied and particles would clump together. To overcome this the spiky kernel designed by Desbrun [19] has a positive gradient everywhere.

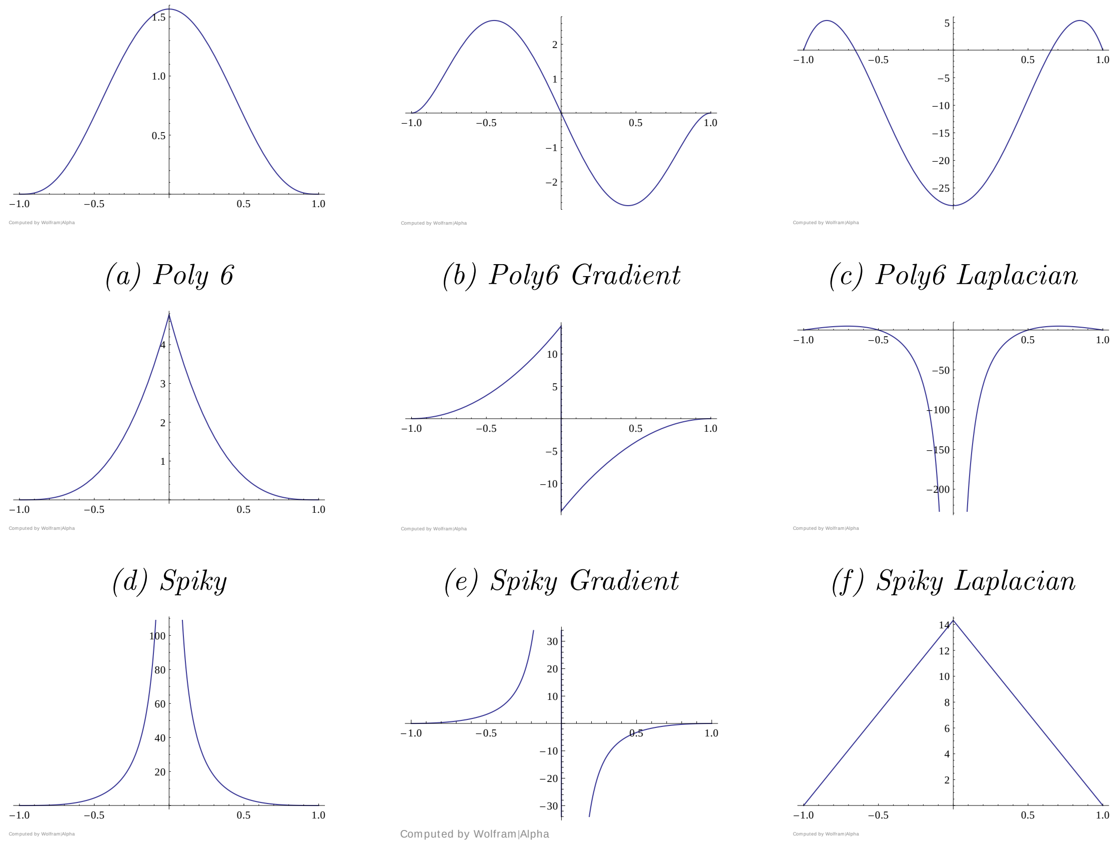
Viscosity Kernel *Laplacian ∇^2 used to smooth $\mathbf{F}^{viscosity}$*

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

$$\nabla W_{\text{viscosity}}(r,h) = \frac{15}{2\pi h^3} r\left(-\frac{3|r|}{2h^3} + \frac{2}{h^2} - \frac{h}{2|r|^3}\right)$$

$$\nabla^2 W_{\text{viscosity}}(r,h) = \frac{45}{\pi h^6}(h - |r|) \tag{4.20}$$

This kernel was also designed by Müller et al. [45] and is applied to the velocity field to smooth viscosity forces. Again referring back to the Navier Stokes terms 4.2 we can see that $\mathbf{F}^{\text{pressure}}$ is the Laplacian (∇^2) of the velocity field. In two dimensions this is simply the second derivative, but if we look at the poly6 and spiky kernels we see that the second derivative becomes negative toward the center. Since viscosity is a frictional force, negative viscosity would imply that particles get sucked together and increase in relative speed as they get closer. This adds energy into the system and produces instability. Instead the viscosity kernel has a laplacian which is positive everywhere and thus as particles get closer frictional forces increase. By using this kernel Müller et al. do not require the additional damping used by Desbrun [19] to overcome this issue.



(a) Poly 6 (b) Poly6 Gradient (c) Poly6 Laplacian

(d) Spiky (e) Spiky Gradient (f) Spiky Laplacian

(g) Viscosity (h) Viscosity Gradient (i) Viscosity Laplacian

Figure 4.3: The 3 specially designed kernels used for SPH simulation, along with their first and second derivatives in 2 dimensions (also referred to as gradient and laplacian respectively). Note how the gradient of the spiky kernel (e) is non-zero at the centre to prevent clumping and how the laplacian of the viscosity kernel (i) is positive everywhere to avoid introducing extra energy through negative viscosity.

Chapter 5

Rendering Theory

5.1 Introduction

Our aim is to produce visually convincing fluid, in order to do this we will need to transform the 3D cloud of SPH particles into a continuous fluid volume and ultimately produce a 2D image from that. One approach is to generate a polygon mesh representing the surface of the fluid, this mesh can then be rasterized and shaded using a standard graphics pipeline such as OpenGL or DirectX. We will discuss the most common algorithm for this, which is Marching Cubes originally proposed by Lorensen [37]. Although Marching Cubes produces visually excellent results it can be too expensive to use in realtime applications, this is because it generates a lot of geometry that is not visible to the camera. Müller [46] presents a technique for generating meshes in screen space using the depth buffer and Marching Squares algorithm.

The above methods are considered indirect volume rendering since they generate a mesh which is later rasterized. It is also possible to generate a final image directly from 3D data, Meissner et al [40] provide a review of popular direct volume rendering techniques framing them in a common theoretical framework and comparing their performance and visual characteristics. Conceptually all methods can be regarded as either object-order or image-order. Splatting is an object-order (or forward-projecting) technique since the algorithm iterates over all particles and projects them forward into the image plane. On the other hand raycasting is an image-order (or backward projecting) technique since the algorithm iterates through all pixels calculating their value by projecting rays back into the scene. These two approaches are depicted in figure 5.1.

We will discuss the details of these two algorithms and mention some techniques available for accelerating them. Unfortunately many of these techniques assume static datasets sampled in a regular grid and therefore use hierarchical subdivision structures. These structures would need to be regenerated each frame as the simulation progresses, which is costly.

In order to give a convincing visual appearance, we need to model some of the optical properties of liquids such as refraction and reflection. Van der Laan [67] present methods

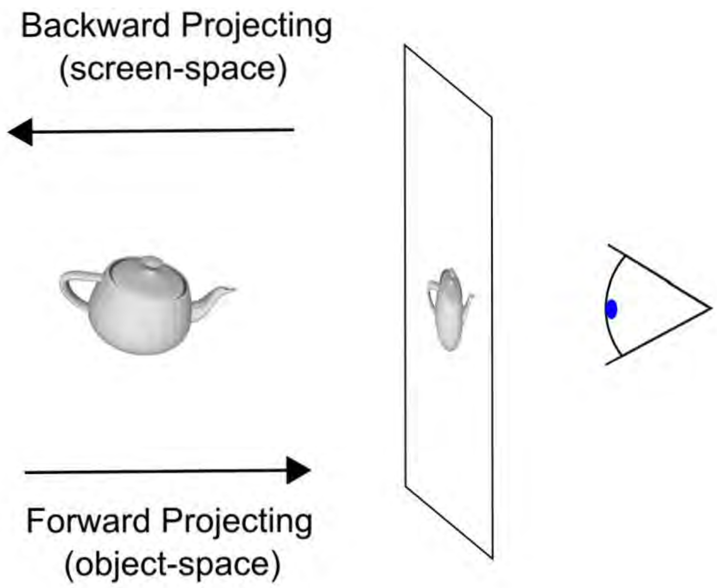


Figure 5.1: Forward vs Backward projection

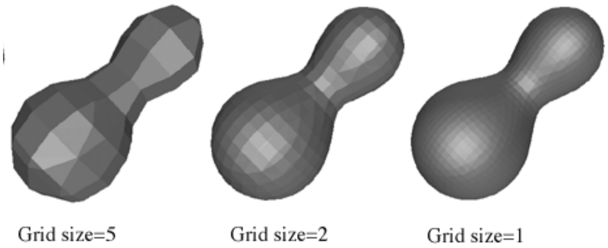
specifically designed for realtime liquid simulations, including convincing liquid effects. Note that these techniques sacrifice physical accuracy for speed,[69] contains a section reviewing algorithms for more realistic offline scenarios.

5.2 Indirect

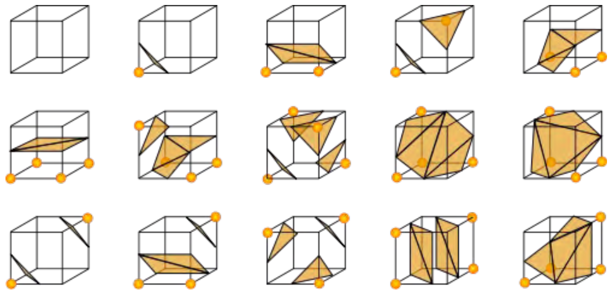
5.2.1 Marching Cubes

The output of the marching cubes algorithm is a polygonal mesh representing the surface of a volume, for our purposes a volume of liquid. This is depicted in figure 5.2a. If we think of the liquid as being defined by its density then we can pick a density value for the fluid. Any region with a density value larger than this is considered fluid while any region with a density value smaller is considered air. The boundary between regions of higher and lower density is referred to as the isosurface, and the transition value between liquid density and air density is referred to as the isovalue. An easy way to understand this would be to visualise the 2D case of an isoline, or contour line, which is a curve drawn on a map to represent line of points at the same height.

If we want to extract an isosurface from a density field the first thing to do is to subdivide divide the space into a 3D voxel grid. Now for each block we look whether each of its neighbours have density values more or less than the isovalue, i.e. are the neighbours inside or outside the surface. If a blocks neighbours are completely outside or inside the surface then no triangles need to be generated, but if some of its neighbours are above and some below then this cube forms part of the surface. To decide what triangles to generate we can construct an 8 bit key based on which neighbours are full or empty. This



(a) Marching Cubes rendered at various grid resolutions. [17]



(b) Triangle lookup table for Marching Cubes algorithm [1].

key will be used in a lookup table to determine which edges are intersected by triangles. Since there are 8 neighbours this table has 256 entries but due to reflection and rotation there are in fact only 15 unique combinations, shown in figure 5.2b. Once we know what triangles to generate for this block the final step is to interpolate the positions of the triangle vertices along the cube edges. By applying this to all cubes in the grid we will generate a polygon mesh.

The main advantage of Marching Cubes is that it fits in neatly with the existing triangle pipeline and can leverage existing shading tools. However a relatively high resolution grid is needed in order to generate visually acceptable results, we can see the effects of this in figure 5.2a. This can mean a lot of wasted memory if the surface only occupies a small volume relative to the full grid (since empty cubes have to be stored and processed). It is also computationally expensive since we have only generated the triangle coordinates so far, they still need to be rasterized. Finally in order to be used for SPH application we need to transform a particle sampled field into a fixed grid. This can be done by counting the number of particles in a grid cell and setting its density accordingly. Fortunately many SPH simulations use a spatial grid to accelerate neighbour particle lookup, so this may not be a problem since the grid is already available. It does, however, mean that rendering resolution is bound to simulation resolution which is not ideal when trying to balance performance and visual quality. For example a coarse grid might be acceptable for simulation but produce visible blockiness when rendering.

5.2.2 Screen Space Meshes

The fundamental performance drawback of Marching Cubes is that much of the triangles making up a surface may not be visible from the current camera position, usually because they are on the opposite side of the volume and facing away. This means a lot of time is wasted producing them. Müller et al address this by generating meshes in screen space [46].

The basic idea is similar to Marching Cubes, but instead of operating in 3 dimensions the algorithm operates on the depth map of the rendered scene. The depth map is used to generate 2D silhouettes of particles, and a procedure similar to Marching Squares is used to generate the mesh in screen space. Finally the screen space mesh is transformed back into world space and rendered.

Since this algorithm only considers particles that are visible to the camera it can operate much faster than Marching Cubes making it suitable for realtime applications. But the resulting mesh is only valid from one viewing direction so it cannot be used for illumination effects like shadows or caustics.

Figure 5.3: Screen space meshes [46]

5.3 Direct

5.3.1 Volume Rendering Integral (VRI)

The VRI is an important concept for understanding both forward and backward projecting algorithms. Let's consider the image to be made up of different coloured light travelling out of the scene and striking a plane. The VRI calculates $I_\lambda(\mathbf{x}, \mathbf{r})$. This is the amount of light of wavelength λ hitting point $\mathbf{x}$ on the image plane from direction $\mathbf{r}$, where $\mathbf{r}$ is usually the direction out from the scene. This can be seen in figure 5.4. Our formulation is based on that of Meissner [40].

$$I_\lambda(\mathbf{x}, \mathbf{r}) = \int_0^L C_\lambda(s)\alpha(s)e^{\left(-\int_0^s \alpha(t)\,\mathrm{d}t\right)}\,\mathrm{d}s \tag{5.1}$$

Here L is the length of the ray, meaning the integral is evaluated along the entire ray. The three factors of the integral are as follows: $C_\lambda(s)$ is the light emitted by point s; $\alpha(s)$ is the opacity at s. The final factor $e^{\left(-\int_0^s \alpha(t)\,\mathrm{d}t\right)}$ represents the attenuation of light as it passes through the intervening material between s and the observer. The closer the point is to the surface the shorter the length of the integral over s is and thus the amount of attenuation is lower.

While the integral form of the VRI describes a continuous function it is not possible, in the general case, to evaluate the integral directly a such. This is because the volume through which the ray moves can have completely arbitrary opacity values, as opposed to functions which yield well to analytic solutions. Since we cannot evaluate the integral directly we need to sample along its path instead, for which we will need to convert it to a discrete summation as follows:

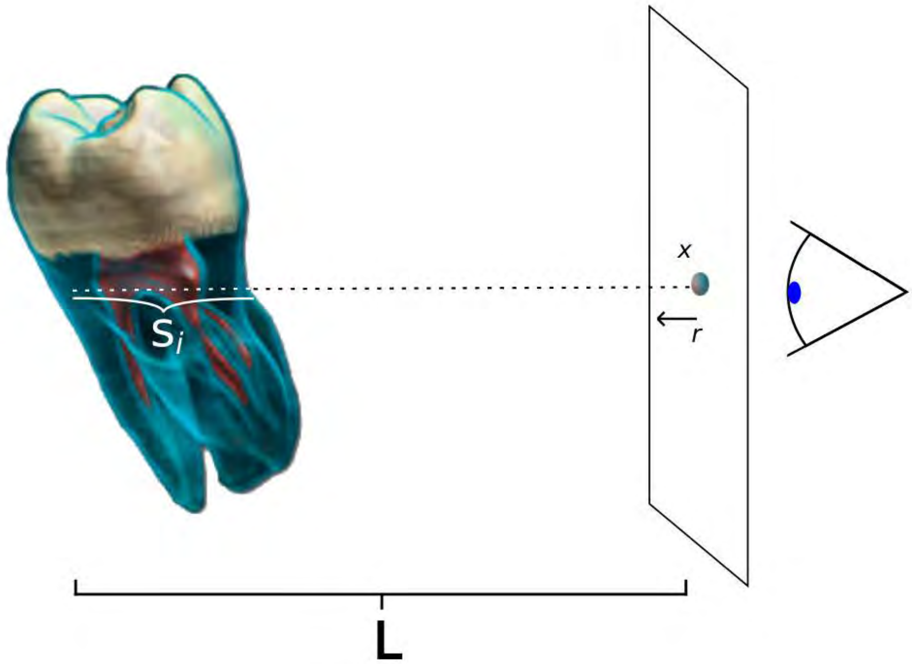


Figure 5.4: Direct Volume Rendering Integral, the dot on the image plane represents the pixel value calculated by stepping along the ray starting at point x in direction r for length L, and sampling the colour and opacity of each data point S_i along the way resulting in a translucent image of the object being viewed. The DVRI can be implemented in either a forward or backward projecting manner.

$$I_\lambda(\mathbf{x}, \mathbf{r}) = \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \cdot \prod_{j=0}^{i-1}(1 - \alpha(s_j)) \tag{5.2}$$

This is now referred to as the Discrete Volume Rendering Integral (DVRI) and the process of accumulating these opacities to form an image is referred to as compositing. We will now see how raycasting and splatting are two different ways of implementing this concept.

5.3.2 Raycasting

Raycasting is an image oriented method, which means that the algorithm iterates through each pixel and calculates its value separately. The value of a pixel is calculated by first casting a ray from its screen-space position into the scene, then sampling opacity values along each ray and compositing them into a final value. This process is visualised in figure 5.5. We can see how this implements the DVRI by stepping down each ray and building up each pixel by sampling opacity values and adding them to a running sum, weighted by the attenuation factor.

A common technique for improving the performance of raycasting is to skip through empty space, unfortunately this requires some form of spatial subdivision structure like an octree which is generally too expensive to regenerate every frame of a simulation.

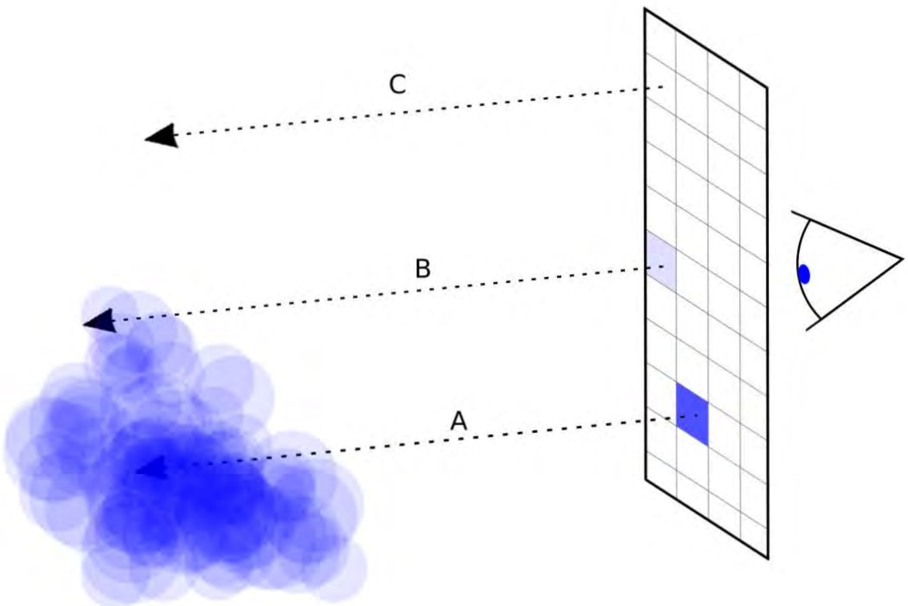


Figure 5.5: Conceptual overview of the raycasting algorithm. Rays are being cast from points A and B and C in order to determine the respective colours of those pixels. As the rays trace into the scene they accumulate opacity while passing through objects in the scene (in this case fluid particles). The ray from pixel A passes through a dense region of fluid and therefore outputs a darker pixel, while the ray for pixel B passes through a relatively sparse region and outputs a lighter pixel. The ray for C passes through empty space and therefore leaves C blank.

Another optimization that can be applied is to terminate rays once a threshold opacity has been reached. This saves iterating through regions which would not be visible the final image.

Since each pixel value is calculate independently, raycasting is ideally suited to parallelization. Krueger and Westermann [35] present a classic method for performing raytracing on the GPU using pixel shaders, later in section 7.2.2 we describe how to use a CUDA kernel to implement the same idea.

An important issue when using to render SPH data is that the algorithm assumes the volume data is continuously defined and can be sampled at any point in space. Since SPH data only generates point positions we need some way of defining values at positions between these points. The naive solution would be to simply re-apply the SPH interpolation function, but that would be fairly time-consuming. Inácio et al [32] handle this by rendering the points into a 3D texture as fuzzy spheres. They then use a method based on that of Kruger and Westermann where sampling along the rays is performed by interpolated texture fetches.

5.3.3 Splatting

Splatting is a forward-projecting, object-order algorithm originally proposed by Westover in [71] to allow interactive exploration of volumetric point-cloud datasets. One of the main motivations for developing the method was to avoid the cost of generating geometry, as required by indirect methods described in Section 5.2. The basic idea behind the algorithm is that each point in a dataset is projected from 3D space onto a 2D image plane, if each point is considered to be a sphere then the resulting image projection will be a circle. This circle is called the footprint and the result of projecting the footprints of all the points onto the image plane will be a 2D representation of the volume. If the volume is not completely opaque then the opacity of individual points can be accumulated to represent varying thickness in the dataset. This entire process is depicted in figure 5.6.

In practice the points will usually be rendered as Gaussian blurred circles, since this avoids sharp visible discontinuities. This can be accomplished by generating a camera aligned plane for each point onto which the resulting blurred circle is rendered. Also the transformation from world space into image space is accomplished using a projection matrix which converts 3D coordinates into 2D coordinates using either a standard orthographic or perspective projection matrix. The details of how splatting can be applied to fluid rendering using are described in more detail in section 7.2.1, where we implement a method described by Van der Laan and Green which uses several splat-based rendering passes to produce the final image of a 3D volume on screen. We make use of the OpenGL API to handle the perspective transformation of the points, to generate the image space footprints and to perform the compositing and blending of the final image.

An optimization to the basic splatting algorithm is surface splatting, which renders only an outer shell and is suitable for datasets such as those captured by laser scanners. Rusinkiewicz and Levoy [59] present a high performance implementation for rendering such datasets, Zwicker et al [79] extends the method to render textures and Botsch et al [15] present a high performance GPU implementation.

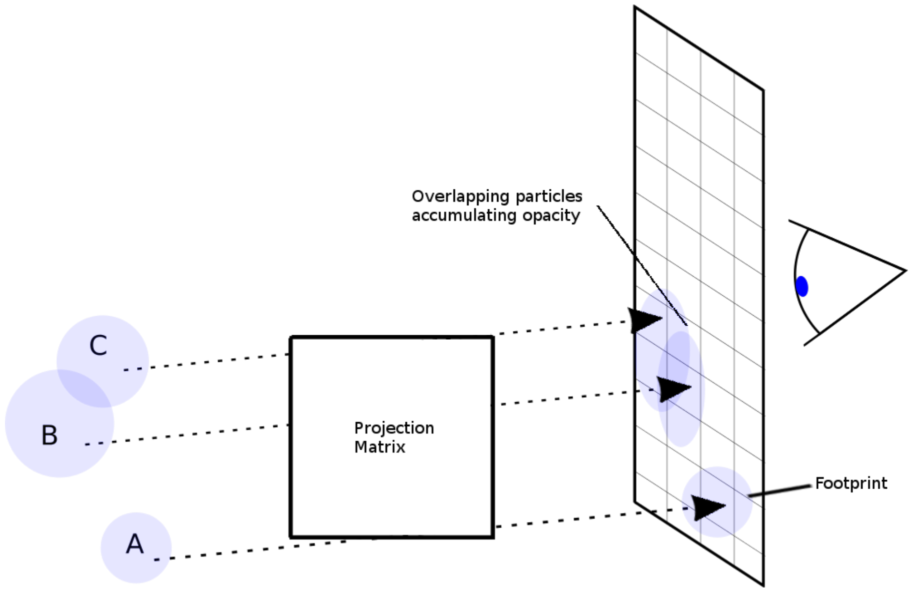
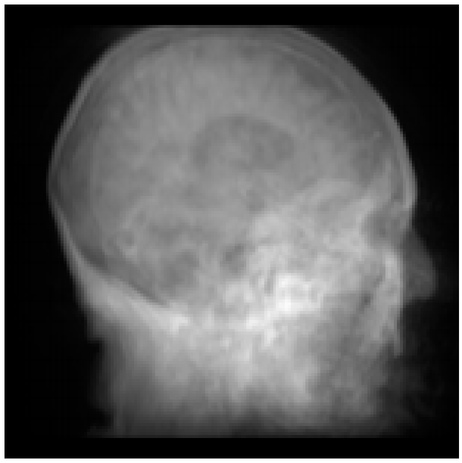


Figure 5.6: Conceptual overview of splatting. Each point on the left projects its footprint onto the image plane, this is performed by multiplying the world coordinates by a projection matrix. The footprints of B and C overlap and therefore accumulate opacity and result in a darker patch (in a real graphics system the entire pixel would be shaded evenly). A is separate and therefore does not accumulate any opacity.

These methods would not be suitable for an SPH simulation though since they would require extracting an iso-surface first. Xue and Crawfis [75] present a GPU accelerated method for rendering volumetric datasets using two different optical models: low-albedo and x-ray. The low-albedo model (figure 5.7a) generates a view similar to flying through a volume slice by slice, to do this the points first need to be sorted in space. Then they are rendered in back-to-front order according to distance from the camera and composited using OpenGL blending. The x-ray model (figure 5.7b) simply renders all points in arbitrary order and averaging them together, meaning they do not have to be spatially sorted. To render a uniform fluid the x-ray model would be sufficient, though depicting different densities or colours would require the low-albedo model.

(a) Low Albedo (Xue and Crawfis [75]))



(b) X-ray (Xue and Crawfis [75])

Figure 5.7: Two different optical models for volume splatting.

5.4 Liquid Optics

So far all the techniques mentioned have generated either opaque surfaces such as figure 5.8a or semi-transparent volumes such as figure 5.8b. In order to create a visually convincing liquid we need to produce several characteristic optical effects of a real liquid volume, these being colour absorption, reflection, refraction and caustics. All of these effects are derived from underlying physical phenomena, but details of their application to rendering specifically are covered more thoroughly by Akenine et al [9].

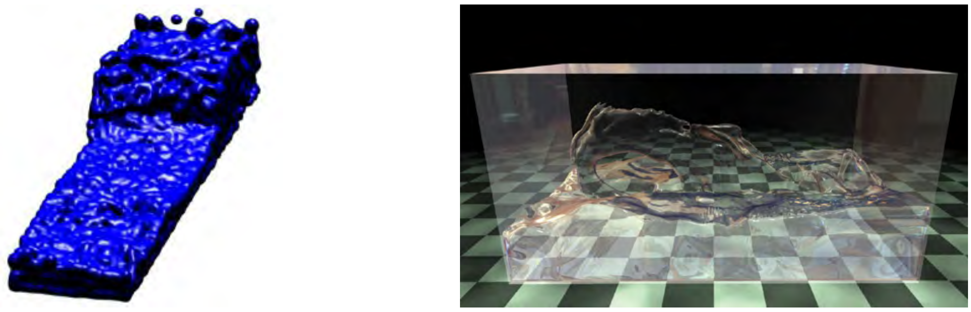
5.4.1 Colour Absorption

The Beer-Lambert Law, commonoly referred to as Beer's Law, is a physical principle which governs the attenuation of light, due to absorption, as it moves through a medium. Given light of a input strength L_{in}, the amount of light L_{out} that is transmitted (i.e. not absorbed) is an inverse function of the distance the light travels in the medium weighted by an attenuation factor. This is expressed mathematically as follows:

$$L_{out} = L_{in} e^{-\alpha d} \tag{5.3}$$

where α is the attenuation coefficient and d is the distance travelled through the medium. Larger values for both of these correspond to more absorption and therefore darker output. The attenuation constant also varies depending on the wavelength of the transmitted light, this corresponds to colours being absorbed differently thereby giving the transmitting medium its apparent colour. In our application the transmitting medium is water which typically appears blue, this means that light of blue wavelengths will not be strongly absorbed (large α) while light of green and red wavelengths will have less absorption (smaller α).

5.4.2 Reflection and Refraction



(a) Opaque fluid surface (from [32]) *(b) A transparent fluid volume (from [53])*

Figure 5.8: Comparison of rendering techniques for liquids, (a) only shows the surface while (b) displays all the optical properties of a liquid such as reflection and refraction.

When light passes between two mediums with different refractive indices, some will be reflected and some will be transmitted. To work out this ratio we use the Fresnel equations, one for light that is polarized perpendicular to the surface plane (s-polarised) and one for light polarised parallel to it (p-polarized). The equations give the reflected light values R_s and R_p.

$$R_s = \left(\frac{n_1 cos\theta_i - n_2 cos\theta_t}{n_1 cos\theta_i + n_2 cos\theta_t} \right)^2 \tag{5.4}$$

$$R_p = \left(\frac{n_1 cos\theta_t - n_2 cos\theta_i}{n_1 cos\theta_t + n_2 cos\theta_i} \right)^2 \tag{5.5}$$

Where n_1 and n_2 are the refractive indices of the two media, and θ_i is the angle of the incoming light. The angle of the reflected light θ_r is the same as the incoming but mirrored around the normal, while the angle of the transmitted light is θ_t and can be worked out using Snells Law.

$$\frac{sin\theta_i}{sin\theta_t} = \frac{n_1}{n_2}$$
$$\theta_t = arcsin(\frac{n_1}{n_2}sin\theta_i) \tag{5.6}$$

For unpolarized light equations 5.4 and 5.5 are combined as follows:

$$R = \frac{1}{2}(R_s + R_p) \tag{5.7}$$

When fully expanded the Fresnel equation becomes rather unwieldy, so Schlick's Approximation can be used instead which is cheaper to evaluate.

$$R = R_0 + (1 - R_0)(1 - cos\theta)^5 \tag{5.8}$$
$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{5.9}$$

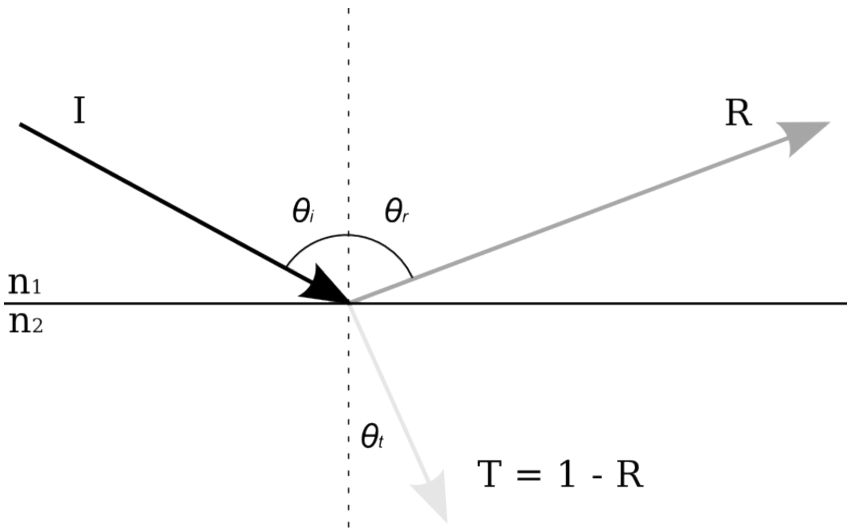


Figure 5.9: Incident light I hits the interface between two media with optical indices n_1 and n_2. The light is partially reflected (R) and partially transmitted (T). The intensity of R is calculated using either the full Fresnel equations (5.4 and 5.5) or the simplified form (5.8 and 5.9), while the intensity of T is simply $1 - R$. The angle of reflection θ_r is the mirror of θ_i, and the angle of the transmitted ray θ_t is determined by Snells Law 5.6.

5.4.3 Caustics

Caustics are a characteristic pattern created when light passes through the surface of a liquid or other medium and can be seen in figure 5.10. The effect is caused by rays of light being refracted as they pass through the uneven surface and thereby focusing along lines which correspond to waves on the surface. We will not be implementing caustics in this work, but more information on how they may be implemented can be found in [74].

Figure 5.10: Caustics caused by the refraction of light passing through the ripples on the surface of a pool of water. These characteristic patterns can add an extra element of visual realism to a scene with water but are beyond the scope of the current work.

Chapter 6

Simulation Implementation

Chapter 4 details the theory of how the Navier Stokes equations describe the motion of a fluid and specifically how Smoothed Particle Hydrodynamics can model those equations using Lagrangian particles. Chapter 3 presents the CUDA programming model which exploits the massively parallel processing power of modern GPUs to provide performance increases for general purpose tasks.This chapter first describes how the CUDA programming model can be use to implement a GPU accelerated SPH simulation, and then how the Bullet physics engine is used to incorporate rigid rigid bodies.

6.1 Data Structures

In order to run the simulation, the physical properties of the particles need to be stored in various buffers. These buffers represent the values required to perform the SPH calculations as shown in Chapter 4 and are listed in table 6.1. Although the simulation is run on the GPU device, they are first initialised on the host and then uploaded, While the host memory is allocated using standard *malloc()*, the memory on the device is allocated using the CUDA library function *cudaMalloc()*. The transfer is performed using *cudaMemcpy()* which is performs much the same function as memcpy().

An important point to note is that the layout of the data has important performance implications. Typically one would declare a struct containing the properties of an individual particle, and then store an array of those structs. This is referred to as an Array of Structs pattern and is depicted in the top row of figure 6.1. The problem with this is that it can pollute the cache with unused data. Recall from Chapter 3 that the GPU has a limited amount of L1 and L2 cache. In a scenario where for example only the density values were being accessed, all of the surrounding pressure, force and velocity values would also be cached since they are alongside the density. A more efficient layout which is recommended by [50] is the Structure of Arrays as depicted in the bottom row of figure 6.1. The values for each property are layed out contiguously, so if only those values are being accessed then better use is made of the cache. One can see in Algorithm 1 that this is the case, with first the density, then the pressure, then the force values each being read by separate kernel launches.

The data stored for each particle is relatively small (roughly 100 bytes), so since the typical GPU contains between 1 and 2 gigabytes of RAM one would expect to be able to carry out simulations on tens of millions of particles. Nevertheless there are also overheads required to store the acceleration grid covered in section 6.4 so the actual figure for the largest scale simulation is expected to be somewhat lower. The details will be discussed in chapter 8.

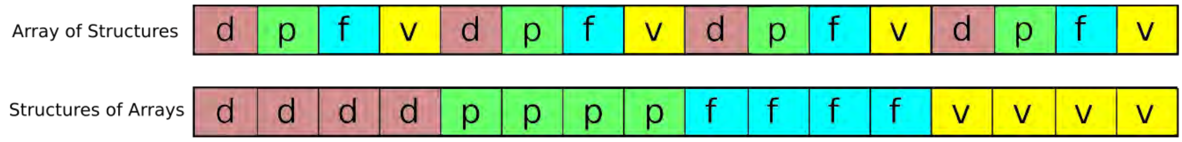


Figure 6.1: The structure of arrays provides better cache performance. When iterating over just the values of a single property the cache is able to load contiguous chunks rather than having other unused values interspersed.

Property	Datatype	Description
`density`	`float`	The fluid density at the position of each particle. This is calculated using equation 4.6 to sum and interpolate the mass contributions of neighbouring particles.
`pressure`	`float`	This is calculated directly from the `pressure` buffer according to equation 4.12. The pressure at a particle position is simply its current density minus the rest density.
`force`	`float3`	This is the nett result of all 3 force terms in the Navier Stokes momentum equation (4.2 and 4.4). The pressure gradient is calculated from the `pressure` buffer using equation 4.14; viscous forces are calculated using the `newVelocity` buffer using equation 4.17 and gravity is a constant.
`oldVelocity`	float3	The *un*-interpolated velocity of the particles from the previous timestep.
`newVelocity`	float3	This is the velocity of the particles from the previous timestep interpolated using leapfrog integration (section 6.3).
`position`	float3	The position of each particle.

Table 6.1: Data buffers required for an SPH simulation.

6.2 SPH Calculations

The very first step in the simulation is computing the density field. The density at each point is calculated from the number and proximity of surrounding particles and is calculated using equation 4.6, which is the SPH interpolation formula. In the CUDA programming model this is implemented by assigning a thread to each particle and having each thread loop over its neighbouring particles. This pattern is referred to as a "gather", since each particle sets its own values based on surrounding ones[1]. The neighbouring particles are distance-tested and only the particles within the support radius actually contribute towards the density summation. A neighbour search grid is used to avoid having to test every other particle, this is described in section 6.4.

Once the density is calculated the pressure field follows directly according to equation 4.12 and is calculated in a similar gather fashion (one thread per particle). Finally the forces for each particle are calculated according the Navier Stokes equation (4.4), again with a gather operation. The pressure gradient term is the pressure field just calculated and the viscosity term is the velocity calculated in the previous frame. Once the forces are calculated the simulation is advanced using a leapfrog integration scheme.

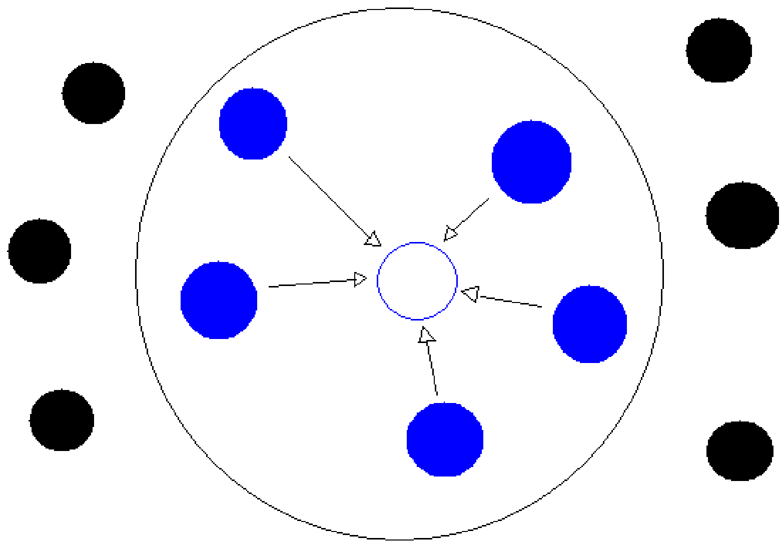


Figure 6.2: SPH interpolation concept: the pressure value of the central particle is calculated by the weighted sum of the particles surrounding it. Only particles within the limited support radius contribute.

6.3 Timestepping

As seen in equation 4.4 the Navier Stokes equations describe the change motion of a particle as a result of the surrounding forces applied to it. With this information we can

[1]The opposite situation would be a "scatter" where every particle contributes weight to its neighbours. This is a very poor parallelisation strategy though since it requires synchronisation on every particle

now advance the particle's position forward in time using an integration. The simplest possible scheme is to calculate each particle's new position based on its current velocity: the velocity is updated based on acceleration forces, multiplied by the timestep size and added to the current position. This is referred to as Forward Euler integration, it is depicted in figure 6.3a and expressed mathematically as follows:

$$\begin{aligned} r(t+1) &= r(t) + v(t)dt \\ v(t+1) &= v(t) + a(t)dt \end{aligned} \tag{6.1}$$

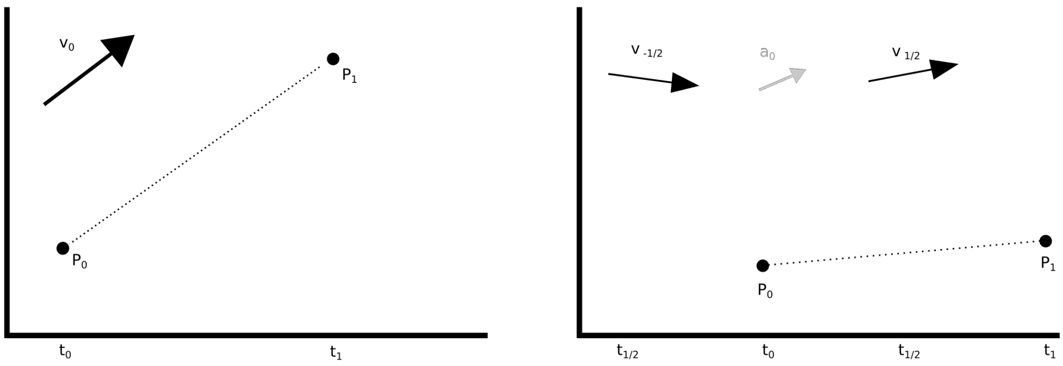
Where v is velocity, r is position and a is acceleration (calculated according to forces such as gravity and pressure). The problem with this procedure is that (since the particle constantly has acceleration forces being applied to it) by the time it reaches its new position at t+1, it is no longer actually travelling at the original velocity $v(t)$. The solution as shown in 6.3b is to interpolate the velocity calculations between timestep positions. This is referred to as Leapfrog integration (since the velocity and position values are stored at half-step offsets to each other) and the mathematical expression is as follows:

$$\begin{aligned} v(t+1/2) &= v(t-1/2) + a(t)dt \\ v(t+1) &= [v(t-1/2) + v(t+1/2)] * 0.5 \\ r(t+1) &= r(t) + v(t+1/2)dt \end{aligned} \tag{6.2}$$

An important factor when considering integration is the size of the error, the Forward Euler scheme is only first-order accurate (which means that the error accumulated in each timestep is proportional to the first power of the size of the timestep), whereas the Leapfrog integrator is second-order accurate (which means the error is proportional to the second power of the timestep size, i.e. the square[2].). Intuitively, what this means is that the Forward Euler scheme is more likely to "explode" if the timestep is too high. Desbrun and Gascuel [19] make use of the Leapfrog integrator for their SPH implementation due to its second-order accuracy. They also list the Courant-Friedrichs-Lewy criterion for ensuring stability, which is essentially that the step size must not be larger than the speed of sound of a wave travelling through the material. Monaghan [44] notes that while a higher order accuracy scheme such as Runge-Kutta 4 might give better accuracy, it lacks the desirable property of preserving angular momentum. This property means the Leapfrog scheme "symplectic", which makes it well suited to molecular dynamics simulations [36].

Using larger timesteps allows the simulation to proceed more rapidly but as the risk of introducing errors. In fluid simulation one of the primary concerns is to preserve incompressibility, essentially this means preventing particles from overlapping each other. As the timestep gets larger the likelihood of this occurring increases, but a predictor-corrector scheme can be introduced which corrects compressibility anomalies introduced.

[2]note the the timestep is assumed to be between 0 and 1, so squaring makes it smaller.



(a) Forward Euler *(b) Leapfrog*

Figure 6.3: In (a) the particle is positioned at P_0 and travelling at v_0 (which was calculated the previous timestep). The current velocity is used to project the particle forward for a full timestep, placing it at position P_1. This would not however be an accurate approximation of its true trajectory since in reality the particle is continuously changing velocity under the influence of forces, thus by time t_1, velocity v_0 is inaccurate. To smooth this the Leapfrog method makes use of velocity values calculated half a timestep previously, thereby interpolating the velocity and reducing the error. In (b) we can see that between $t_-1/2$ and $t_1/2$ the influence of acceleration a_1 has been taken into account, thus the particle is moved by the smoothed velocity.

Solenthaler and Pajarola [63] present a method called PCISPH (Predictive Corrective SPH) which uses a solver in between timesteps to adjust particle positions and preserve incompressibility. Since the solver does not need to find neighbours or compute forces it can run far more cheaply than the simulation step. Significant speedups are reported due to increased timestep size.

6.4 Neighbour Search

As mentioned in section 4.4 the SPH smoothing kernels have limited support radius. This is implemented by having each CUDA thread perform distance checks on neighbouring particles. In principle this distance test could be applied in a brute-force fashion to all particles in the simulation domain, but this would be an unnecessarily expensive $O(n^2)$ procedure. It is therefore desirable to limit the number of candidate particles that are tested, and this is done by using a data structure which allows fast querying of which particles are near others. This is depicted in figure 6.4.

There are three major considerations to take into account when selecting such an acceleration structure: construction cost, query cost and memory cost. Since particles are constantly moving relative to each other this structure needs to be rebuilt every frame, and since our focus is on realtime performance the key factors are construction and query cost. The two major options are hierarchical (such as a k-d tree) or fixed grid. Harada et al [27] note that the construction of a hierarchical data structure is typically $O(n\log n)$ whilst the query cost is $O(\log n)$. On the other hand a fixed-grid allows $O(n)$ construction and $O(1)$ query cost, making it more desirable for realtime applications to moving particles. The downside however is that fixed grids can waste memory by needing to reserve space for empty voxels. A hierarchical data structure is thus more memory efficient but better suited for fixed scenes which only need to be processed once instead of repeatedly like particle data. Zhou et al [77] present GPU optimized a k-d tree construction algorithm and Thrane provides a review of acceleration structure options [65], both in the context of static scene rendering.

A third option is that of hash grids [49], which have a low memory footprint and fast access and construction times. These are not, however, well suited to GPU implementation since hash collisions result in uneven access times. Due to the SIMT architecture described in chapter 3, if one thread in a warp were to require collision resolution then all the others would have to wait for it, thereby losing the amortized linear access time.

We will be using the simple grid structure as described by Green [26] and modified by Hoetzlein [29]. The conceptual structure of the lookup table is depicted in figure 6.5. More sophisticated refinements to the basic fixed-grid are presented by Harada et al [27] (a texture-based, sliced data-structure) and Goswami et al [25] (using shared memory and z-indexing to improve cache performance), but exploring these is beyond the scope of this project.

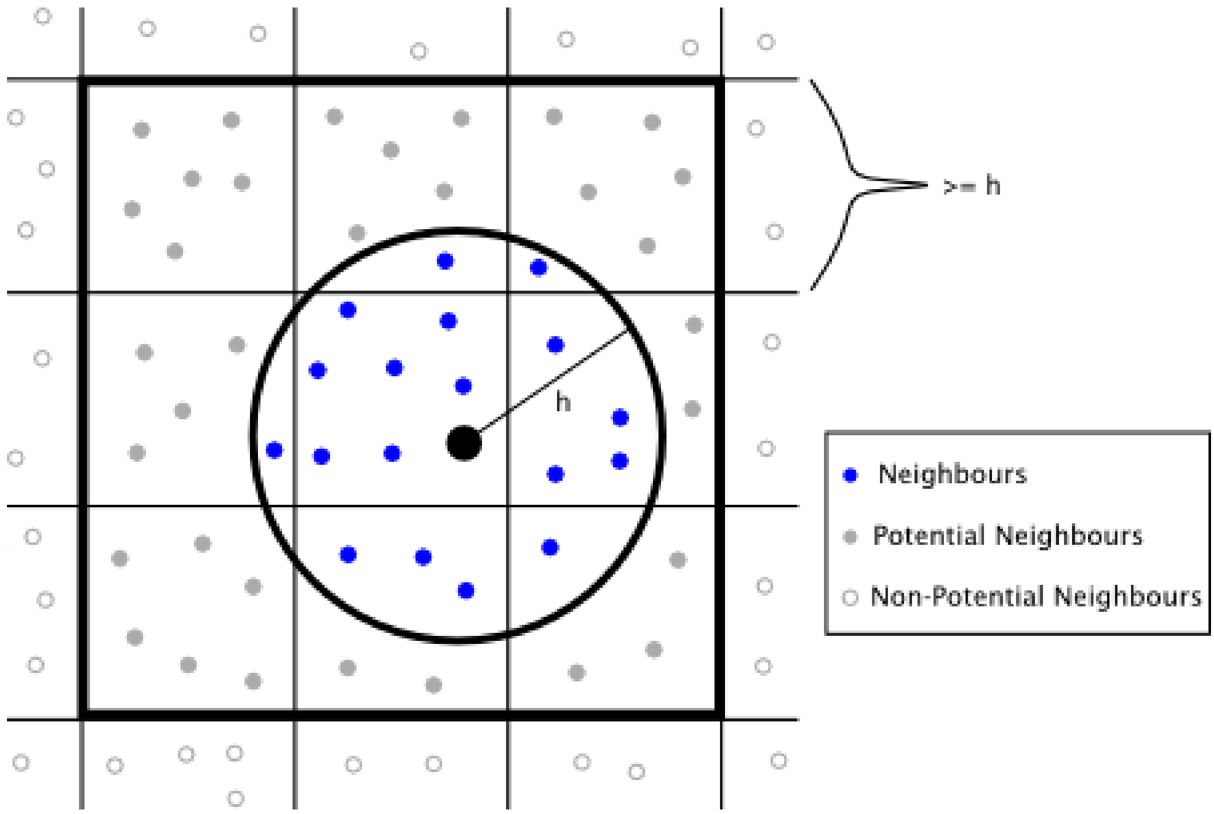


Figure 6.4: Neighbour-search grid. When calculating which particles fall within the support radius (h), only those in adjacent grid cells will be tested (outlined by the thick box). Note that the width of the grid cells has to be greater than or equal to the support radius to avoid missing particles that lies in non-adjacent cells.

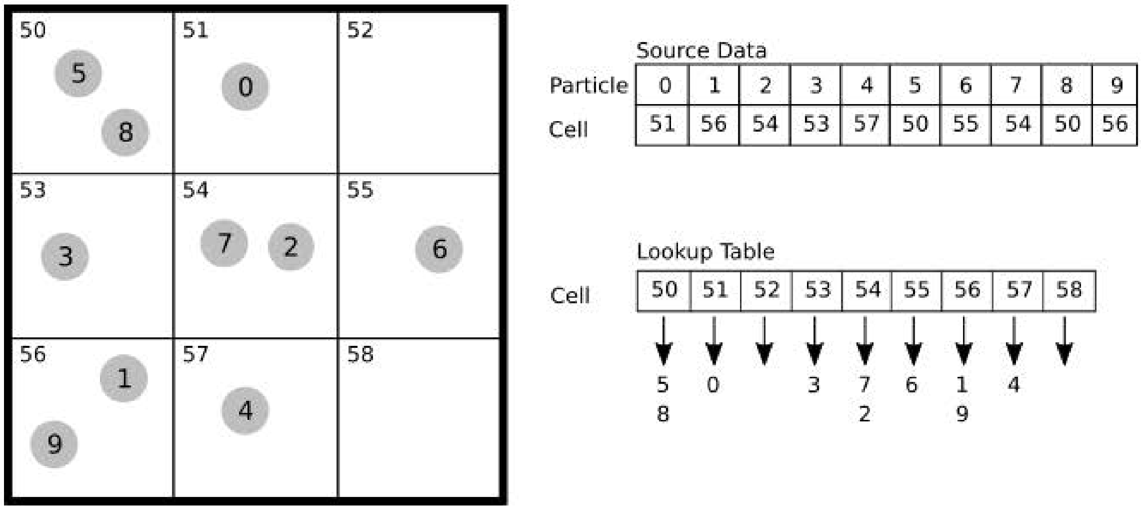


Figure 6.5

6.5 Synchronisation

The CUDA kernels launch asynchronously on the GPU device, which means that execution on the CPU continues immediately after launching the kernel. Since the GPU can only execute one kernel at a time this will cause an immediate crash, but a more insidious bug occurs during the memory transfers. The copy operations on line 6 and 10 also execute asynchronously which means that the CPU may read memory from a host buffer before it has been updated from the GPU thus producing inconsistent results.

Both of the above problems are rectified using `cudaDeviceSynchronize()`. This is a barrier function that halts execution until all device operations are complete. In this situation several kernels are launched consecutively and the memory on host and device is being accessed immediately after it is transferred. In these cases `cudaDeviceSynchronize()` must be called immediately after initiating a device operation. But, if the CPU had separate work to continue with while the GPU is running then the barrier can be moved further ahead thus allowing CPU and GPU to interleave execution and gain a performance benefit.

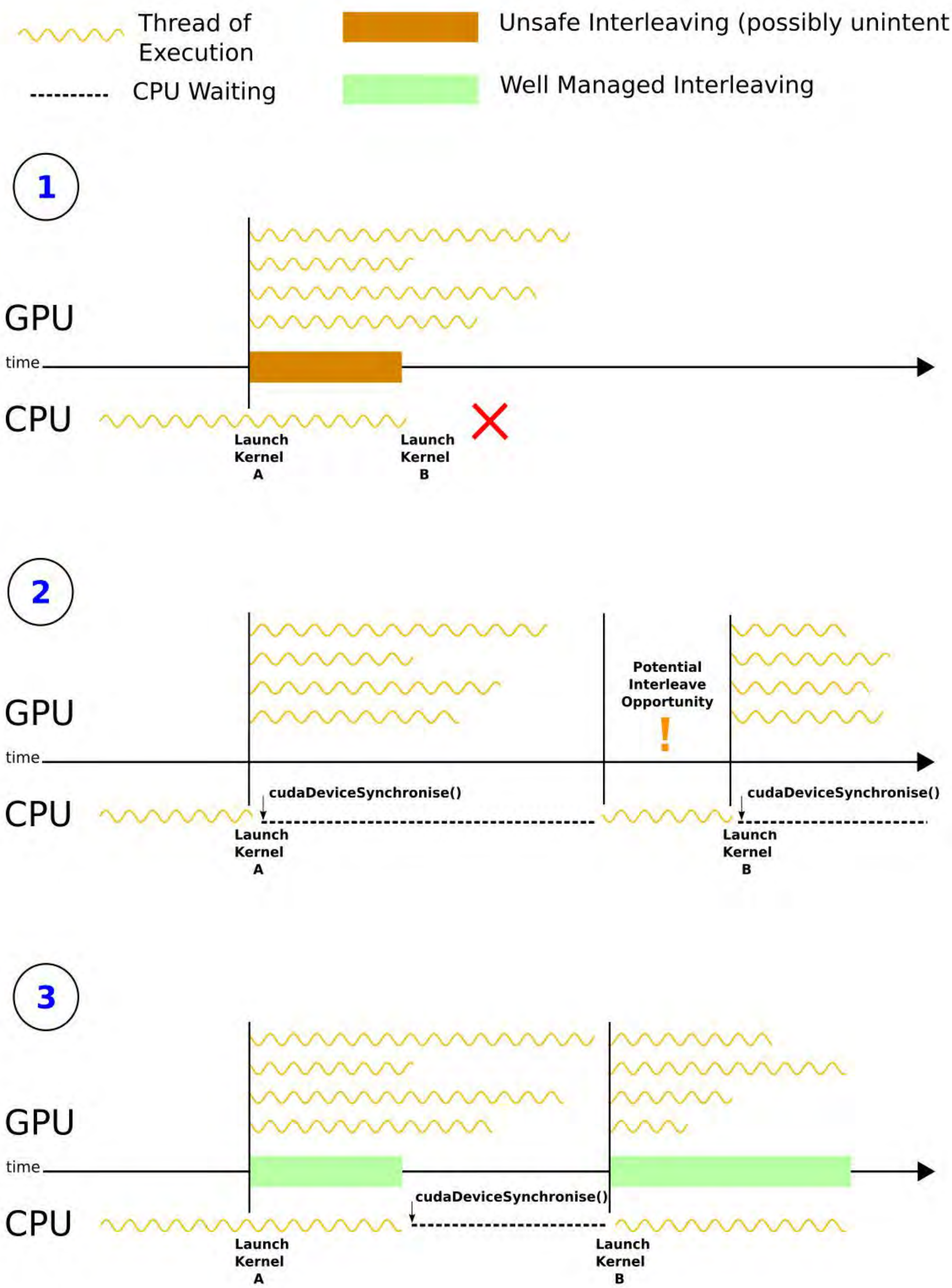


Figure 6.6: Synchronisation of GPU and CPU operations, note that CUDA kernels are launched asynchronously. In case (1) the CPU launches kernel A and continues execution while it runs. Later on the CPU attempts to launch kernel B, but since A is still running this produces an error (only one kernel can run at a time). Case (2) is the opposite extreme, kernel A is launched and the CPU waits for it to complete. But once A is complete there is still some work to to do before launching kernel B. Case (3) is ideal since the CPU proceeds with other work while the kernel is running, and only waits at the point where it wants to launch another. Note how in Case (3) the second kernel is able to launch sooner than it would in Case (2), thereby reducing overall execution time.

Algorithm 1 Simulation Outline

```
for each frame do
    update neighbour search grid

    ComputeFluidPressureField( )                          ▷ CUDA Kernel
    ComputeFluidForces( )                                 ▷ CUDA Kernel
    AdvanceFluidParticlePositions( )                      ▷ CUDA Kernel

    TransferSolidParticleForces( )                        ▷ device to host
    apply forces to bodies
    advance body positions (Bullet Physics)
    update solid particle positions
    TransferSolidParticlePositions( )                     ▷ host to device
```

$r \to$ position; $m \to$ mass; $\rho \to$ density; $P \to$ pressure;
$F \to$ force; $v \to$ velocity; $a \to$ acceleration; $t \to$ time;

See section 4.4 for smoothing kernels.

```
function ComputeFluidPressureField
    for each particle i (in a separate thread) do
        for each neighbour j do
            r = r_i − r_j
            if r ≤ h then
                ρ_i += m_j ∗ Poly6Kernel(r, h)            ▷ eq4.7
        P_i = ρ_i − ρ_rest                                ▷ eq4.12

function ComputeFluidForces
    for each particle i (in a separate thread) do
        for each neighbour j do
            r = r_i − r_j
            if r ≤ h then
                F_pressure +=                             ▷ eq4.14
                    m_j ∗ ((P_i + P_j)/(2 ∗ ρ_j)) ∗ GradientSpikyKern(r, h)

                F_viscosity +=                            ▷ eq4.17
                    μ ∗ ((v_i − v_j)/ρ_j) ∗ LaplacianViscKern(r, h)

                F_i = F_pressure + F_viscosity + F_external

function AdvanceFluidParticlePositions
    for each particle i (in a separate thread) do
        a_i = F_i/m_i                                     ▷ eq 4.4
        v_i += a_i
        r_i += v_i                                        ▷ eq 6.2
```

6.6 Rigid Bodies

The implementation of the rigid bodies is based on that described by Akinci in [11]. The bodies like the fluid are made up of particles allowing arbitrary shaped to be easily defined. Each solid particle is set at a fixed local offset from its parent body's centre of mass, and they execute the same SPH pressure and force calculations as the fluid particles when steps 3 and 4 are run in algorithm 1. Since the inter-particle force calculations are the same this allows two-way interaction with the solids particles to be easily integrated, so the fluid influences the bodies *and* bodies displace the fluid.

However, rather than advancing according to the Navier Stokes equations in step 5 the motion of the rigid bodies is controlled by integrating the Bullet Physics engine. Unless it is applied directly to the centre of mass, any force upon a rigid body will result in both rotational and translational displacement. This is because such a force results in torque being applied around the centre of mass. This torque is simply the cross product of the applied force and the offset from the centre. Therefore, in order to calculate the total resultant torque applied to the body we need to iterate through all the solid particles and accumulate the cross products of their offsets and the forces applied by surrounding fluid particles. This is described in more detail in figure 6.7.

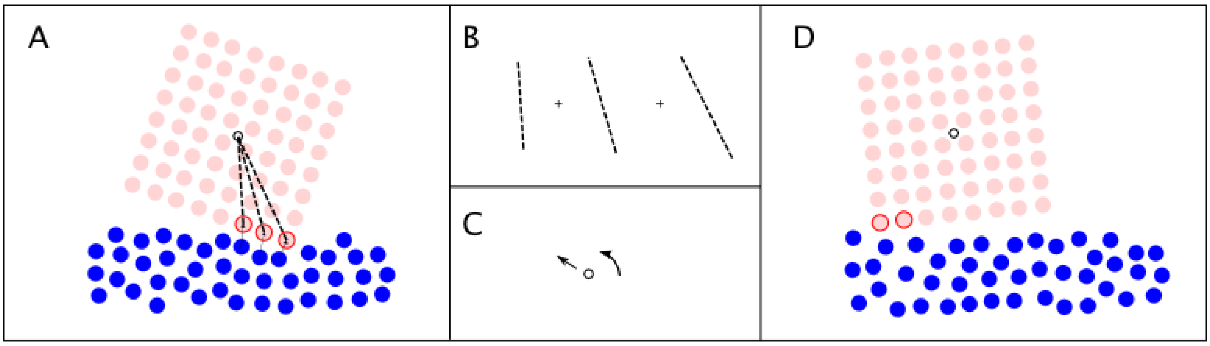


Figure 6.7: An outline of the rigid body interaction. In step A some of the solid particles are in contact with the water, each of these applies a small force at an offset from the centre of mass. In step B the forces from the individual particles interactions are accumulated resulting in both a linear force and an angular force due to torque (step C). This force is the result of the cross product of each particles offset and the force on that particle. Once the total resultant force is calculated it applied to the rigid body, integration is performed by Bullet, a new position is calculated for the next frame (step D) resulting in new solid-fluid contacts. The particles making up the solid all store their local offsets from the centre of mass and their global positions are also updated in step D.

Chapter 7

Rendering Implementation

Chapter 5 describes the different fundamental approaches to volume rendering: direct and indirect. As noted by Van der Laan [67], indirect rendering is ill-suited to realtime applications due to the expense of generating a mesh, especially at high enough resolution to avoid artefacts. Therefore, since the focus of this thesis is realtime simulation, only two direct rendering methods will be explored. Both are also well suited to implementation using the OpenGL programmable pipeline.

Van der Laan et al. [67] present a splatting-type method, entitled Screen Space Fluids, for rendering convincing fluids directly from particle data. The method operates at interactive framerates and reproduces optical effects such as reflection and refraction (as described in Section 5.4) reasonably well. Krueger et al [35] present a fully GPU based raytracing technique, although they focus on volumetric datasets rather than fluid rendering. The benefit of raytracing here is that it is straightforward to incorporate solids within the fluid, whereas with Screen Space Fluids this would be difficult.

7.1 OpenGL Pipeline

OpenGL is a 3D graphics API which renders images to screen from data input to the GPU. Fundamentally three types of data are required to specify a scene: the positions of vertices, the connections between those vertices (forming primitives) and the colour of each fragment. Rendering a scene proceeds in several stages which are applied to this data, each passing its output as input to the next stage. The entire process is referred to as a pipeline, and is depicted in figure 7.1. First each of the vertices are transformed into viewspace by applying the modelview and projection matrices, this can be done using either perspective or orthographic projection modes which determines whether the scene will be rendered like a real camera or like a flattened blueprint. Next primitives are formed and rasterized and finally the colour value for each fragment within the primitves is calculated.

The fragment is not necessarily the final colour that will appear on screen, and can be thought of as a “potential pixel”. The final pixel value is also determined by blending and clipping. As and example: if two opaque shapes overlap, the fragments from the backmost will not be rendered as part of the image while if two transparent shapes overlap their fragments may be blended together to form a final pixel value. Also, a fragment may fall outside of the view frustum of the camera and therefore never be rendered as a pixel at all. Calculating the colours of fragments is called *shading* and is determined either by colours specified at the vertices or by sampling from a texture. In both cases the value is modulated by the lights within the scene, and the relative angle of the surface normal to them.

The way the vertices are transformed as well as the primitive generation and shading model all correspond roughly to real-world materials and objects. They are the default rendering model provided by OpenGL, and together they form what is now referred to as the "fixed-functionality" pipeline and some examples of its output can be seen in figure 7.2. In early versions of OpengGL, this pipeline was all that was available and while many aspects of it could be configured, it could not fundamentally be changed. This fixed functionality was built into the hardware itself, but from OpenGL 2.0 onwards GPU hardware became fully programmable, which meant that arbitrary programs could be executed for each of these stages. These programs are written in a C-like language called GLSL and, since shading stage became programmable first, they are called *shaders*, even when applied to the vertex transformation and geometry stages.

When a draw call is made multiple instances of the shader are launched in parallel, one for each vertex, primitive or fragment depending on which stage is executing. Each instance receives input from the previous stage and outputs to the next stage. Parameters can also be passed to the shaders from the host, these can be in two forms: uniform variables and vertex attributes. Uniform variables are the same for every instance and can be used to pass global values such as light direction and colour. Vertex attributes are passed in arrays and are unique to each vertex, they can carry values such as position, or normal.

Shaders are also able to sample from textures, which are very important in OpenGL. Texturing geometry is a very specialized task since the underlying image is usually filtered or uses mip-maps to handle varying levels of detail. GPUs therefore have dedicated texturing hardware which are called *texture units* in OpenGL. These perform mip-mapping and filtering and also store textures in memory layouts which optimize coherent access for spatial locality [1]. These texture units can be accessed in GLSL where they are referred to as *samplers*. There are 1D, 2D and 3D samplers available.

More details about GLSL can be found in [58], commonly referred to as the Orange Book.

[1] The functionality of the texture units is also available in the CUDA API, see section 3.3.2.

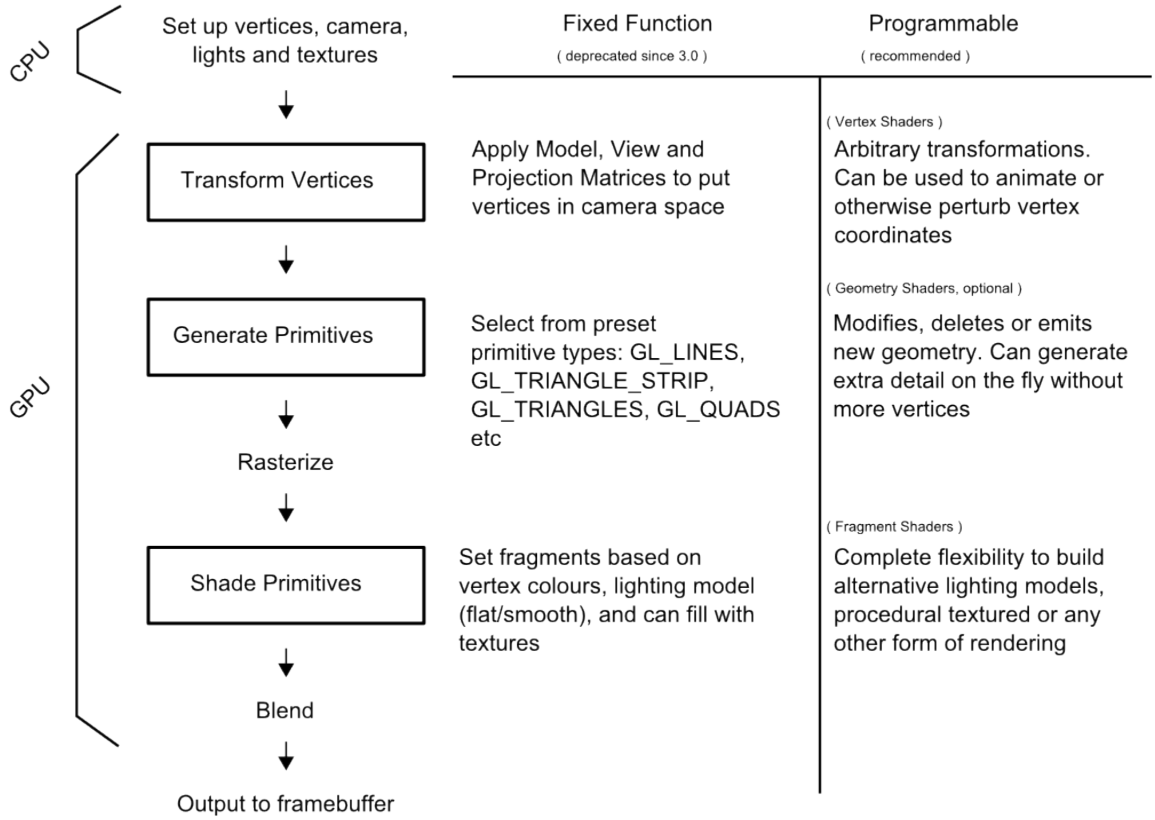
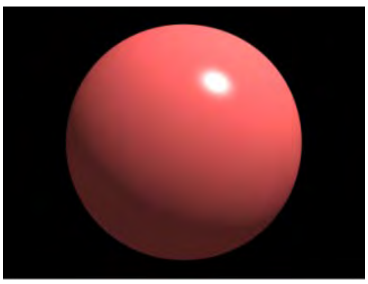
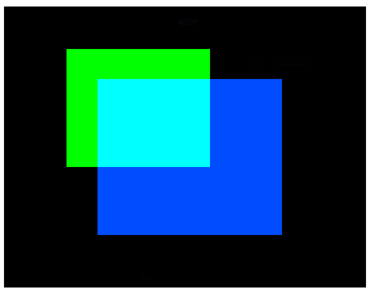


Figure 7.1: A simplified view of the OpenGL rendering pipeline. The boxed steps were initially provided by the fixed functionality pipeline, but have since been replaced by fully programmable shaders. An application making use of the programmable pipeline is required to provide vertex and fragment shaders for every draw call, and may optionally provide a geometry shader.

(a) Texturing



(b) Lighting



(c) Blending.

Figure 7.2: Some of the functionality provided by the original OpenGL fixed pipeline. Texturing fills in primitive shapes with colours sampled from an image, example (a) shows a textured cube. Example (b) shows smooth shading which interpolates the light value at each vertex across the surface of each primitive (as opposed to flat shading which applies the same lighting value across the whole surface). Multiple lights and coloured lights are also supported. Finally example (c) shows two shaded shapes with their colours additively blended together.

7.1.1 Framebuffers

All rendering in OpenGL is output to a memory structure called a Framebuffer. While the windowing system provides a Default Framebuffer whose output is displayed directly onto the screen, it is also possible to render to an off-screen location known as a Framebuffer Object (FBO). In OpenGL the Framebuffer Object is just a container descriptor, so after creation it does not have any memory allocated to store images. In order to render to an FBO we need to attach buffers where pixel data will be stored. An FBO has a depth attachment point, a stencil attachment point (which is used to crop the image) and multiple colour attachment points. If the rendered results are going to be sampled from then we need to attach textures to the colour points since they store data in an optimized layout, otherwise if the images are going to be used directly then we can attach Renderbuffers instead. Our splatting implementation uses several passes that require shaders in later stages to sample from previous stages, thus we use textures in our framebuffers.

7.1.2 CUDA and OpenGL Interoperability

Since both APIs deal with the GPU they are able to interoperate effectively, and this functionality is described in the CUDA Programming Guide [50]. In order to access OpenGL resources from within CUDA we need to initialise the program using *cudaGLSetGLDevice()* and then register each resource we wish to share. The OpenGL resources that may be mapped into CUDA are buffer objects, textures and renderbuffers.

Buffer objects are registered using *cudaGraphicsGLRegisterBuffer()* and appear as a device pointer that can be passed to a kernel and accessed directly as linear blocks of memory using array syntax. Our splatting implementation populates a vertex buffer object in such a manner. Textures and renderbuffers are registered using *cudaGraphicsGLRegisterImage()* and we make use of a 3D texture for our reycasting implementation. Textures are slightly more complicated to handle due to the special texture memory present on the GPU (see section 3.3.2 for details). Since textures are not stored as linear arrays they need to be accessed via CUDA arrays, which are opaque handles to textures which are accessed via functions. Textures that are only read from are stored as *texture* objects, but textured which are to be written to are referred to a *surface* objects. Texture reads are addressed using normalized texture coordinates from 0 to 1, but writing to surfaces requires calculating the actual byte address offsets of the underlying texture elements. For raycasting we output the simulation results to a 32bit floating point RGBA texture. We also need to ensure that we register the surface object with the *cudaGraphicsRegisterFlagsSurfaceLoadStore*.

One final point to note is that OpenGL and CUDA cannot access resources at the same time. Since the OpenGL driver is allowed to shuffle memory for performance reasons it may invalidate pointers that CUDA held. For this reason we need to wrap access to shared resources using *cudaGraphicsMapResources()* and *cudaGraphicsUnmapResources()* and avoid accessing it from OpenGL for that duration. While it is mapped we refresh the handle to the resource using *cudaGraphicsResourceGetMappedPointer()* for buffers and *cudaGraphicsSubResourceGetMappedArray()* for arrays. Similarly, while the resource is not mapped we cannot access it from CUDA.

Algorithm 2 CUDA and OpenGL interoperability. The VBO and texture resources are generated using standard OpenGL calls and registered with CUDA at the beginning of the program. Output is done by a kernel which copies data into the resources from buffers in CUDA global memory space. Note that the pointers are refreshed every frame and that the rendering is performed only after unmapping the resources, since accessing them from OpenGL while they are mapped into CUDA is undefined behaviour.

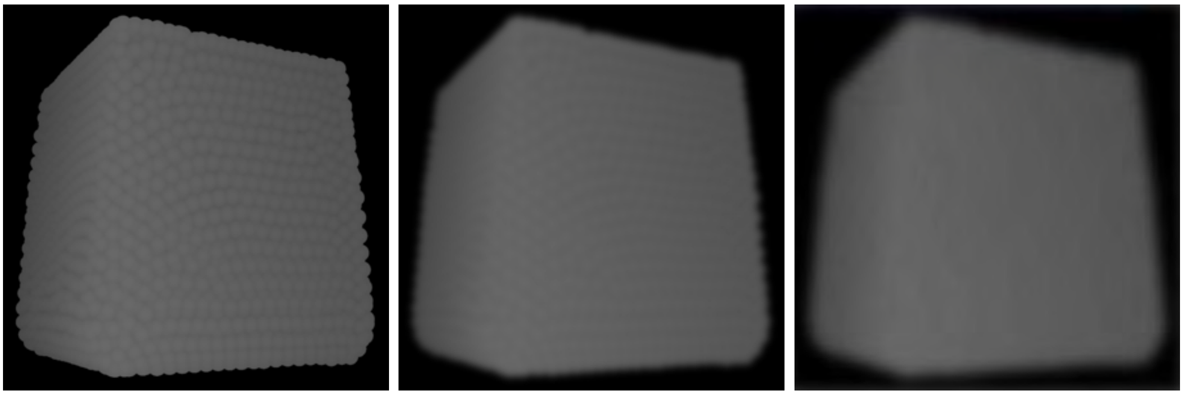
Splattting Mode	RayTracing Mode
glGenBuffers()	glGenTextures()
cudaGraphicsGLRegisterBuffer()	cudaGraphicsGLRegisterImage()
for each frame **do**	**for** each frame **do**
perform simulation in CUDA	perform simulation in CUDA
cudaGraphicsMapResources()	cudaGraphicsMapResources()
cudaGraphicsGetMappedPointer()	cudaGraphicsGetMappedArray()
output results to VBO	output results to the surface
cudaGraphicsUnmapResources()	cudaGraphicsUnmapResources()
Render using OpenGL	Render using OpenGL

7.2 Implementation Details

7.2.1 Splatting

Screen Space Fluid Rendering, presented by Van der Laan and Green in [67], provides a method suitable for real-time rendering of fluids and, although not explicitly stated, is most applicable in the area of games and other interactive applications, which is the focus of our current work. We have implemented the basic ideas presented in the paper, but left out some features which can be addressed in future work. Specifically a major issue with splatting based methods is to smooth the particles so they do not look blobby. The simplest approach is to use Gaussian smoothing while Van der Laan and Green make use of an image-processing technique called curvature-flow which the describe as being analogous to the natural physical phenomenon of surface tension. This yields significant performance benefits over blurring, but we have only implemented blurring since it is simpler. We may explore curvature flow in future work.

Our implementation uses offscreen rendering to produce several separate passes, which are later composited into the final image. The particle positions are transferred to a Vertex Buffer Object (VBO) from where they are rendered as GL_POINTS using the OpenGL glDrawArrays() method. The points are rendered as sprites, which are camera-oriented quadrilaterals. Rather than needing to pass 4 corners per quad the OpenGL point sprite extension generates sprites from single point coordinates, this is a standard extension enabled by default in 3.2 core profile. The shader used to shade the sprites produces a sphere by colouring based on the distance from the center, the point nearest to the camera is lightest, getting darker as it moves outwards. We can see in figure 7.3a that this produces a very bumpy surface, but figure 7.3c shows how this can be corrected using gaussian blurring.

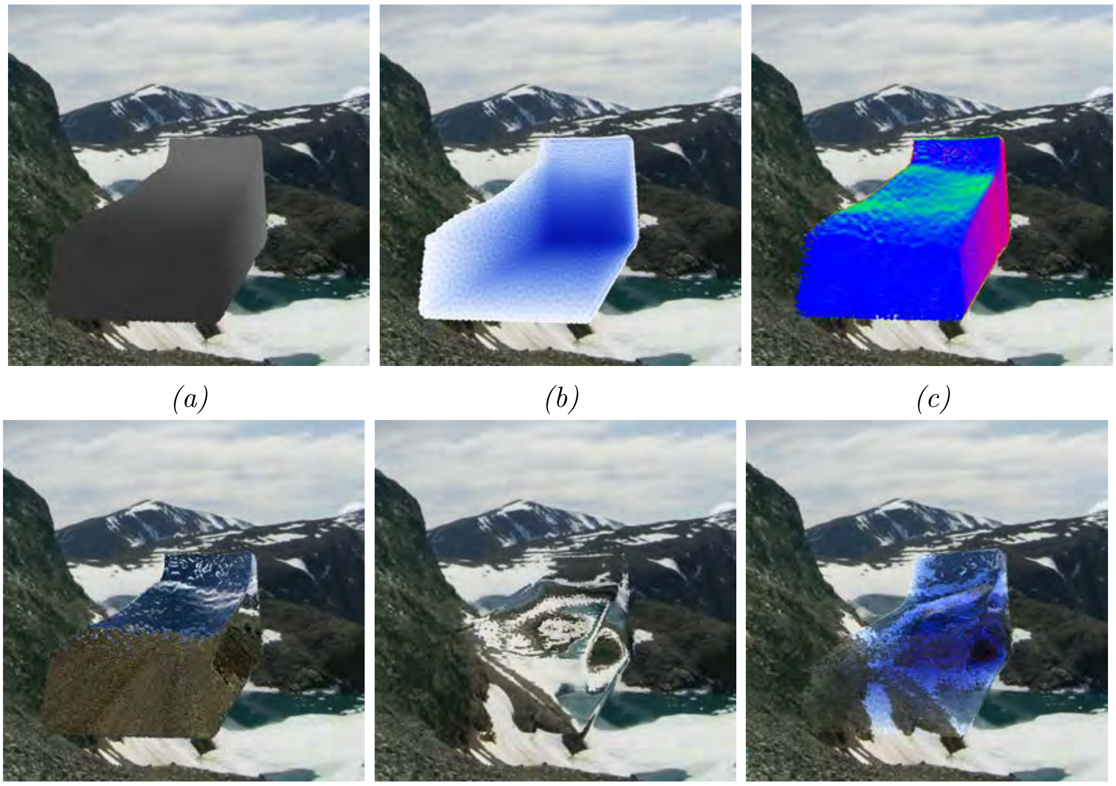


(a) No blurring *(b) Light Blurring* *(c) Heavy Blurring*

Figure 7.3: Particles are rendered as spheres, which produces visible bumpiness as seen on the left. Slight gaussian smoothing can reduce the visibility of the bumps as seen in the middle. Heavy blurring can eliminate the bumps completely as seen on the right but is expensive to perform.

The first two passes render a depth map and a thickness map to separate FBOs by rendering the particles first with depth testing enabled and then with is disabled. The thickness pass is performed using additive blending. This produces an image of the surface of the fluid as shown in figure 7.4a and the thickness shown in figure 7.4b. Using a smoothed version of the depth map, the surface normals are calculated based on the gradient, this is seen in figure 7.4c.

The textures containing the normal and thickness maps are then passed as samplers to the final shader, along with the background view from the OpenGL cubemap. The reflection and refraction are calculated from the normal map and shown in figures 7.4d and 7.4d. At this stage the surface normals are in eye-space, multiplying them by the built-in GLSL variable gl_NormalMatrix transforms them into worldspace. The refraction is achieved by taking a sample from the background scene and offsetting it in the direction of the surface normal. The amount of offset is weighted by the thickness of the fluid at that point and the colour is also attenuated based on the thickness according to Beer's Law. Reflection is performed by calculating a viewing vector for each point, reflecting that around the normal at that point and then sampling from the cubemap. This reflection value is weighted by the Fresnel equation, with shallower angles of incidence reflecting more. The final result is shown in figure 7.4f.



(a) (b) (c)

(d) (e) (f)

Figure 7.4: The separate passes making up the final image: a) depth map b) thickness map c) normal map d) reflection e) refraction f) final.

7.2.2 Raycasting

Our raycasting implementation is based on the idea presented by Krueger and Westermann [35] with modifications to support liquid optical effects, as described in section 5.4. The main difference is that Krueger and Westermann implement their solution using fragment shaders and accomplish looping by using multiple passes rendering into textures. Since our solution is implemented using CUDA we have access to full programmability and can loop within our kernels.

Our raycasting rendering proceeds in two major phases, first the particle data needs to be transferred into a 3D texture and then raycasting is performed on this texture. The particle data resides in GPU memory for the simulation step, and since the rendering will also take place on the GPU it can be transferred directly from CUDA memory into OpenGL memory without transferring back and forth to the host. Note that CUDA and OpenGL each control their own memory on the device, so interoperating between them requires some coordination which we shall describe below.

Recall from section 6.1 that particle data is stored contiguously in a buffer, which means there is no direct way to map an array of positions into the 3D texture. Instead we use a CUDA kernel with each thread mapped to a texture cell, so for example a 128x128x128 texture will launch 2,097,152 threads. The texture coordinates of each grid cell are then mapped into the world space coordinates of the simulation and the fluid field is sampled using the acceleration grid to find the density at each point, calculated using the same gather kernel but rather than interpolating the density at the position of each particle it interpolates at regular intervals defined by the texture dimensions.

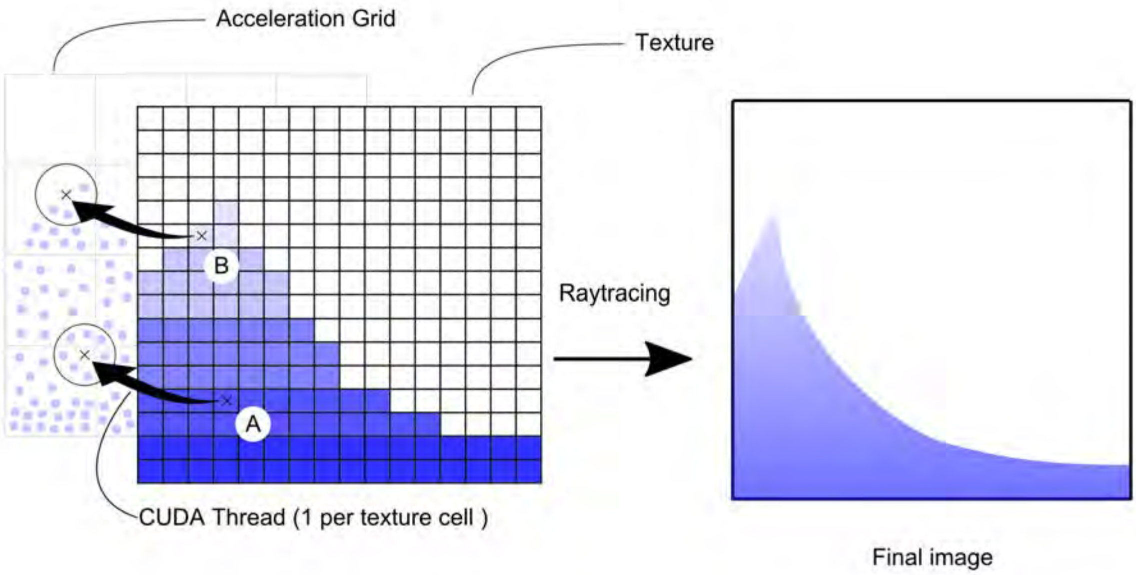
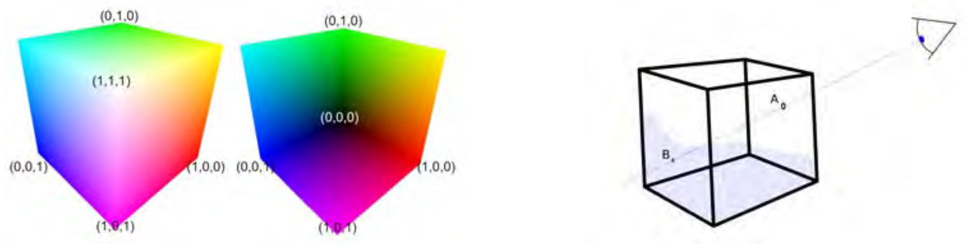


Figure 7.5: Populating a texture using 1 CUDA thread per cell to sample the SPH acceleration grid at evenly spaced intervals. In this case we depict a 2D version of the 3D problem. A and B in the figure are individual cells in the texture, each of which is populated by a single CUDA thread which samples the simulation space at a location corresponding to the cells relative coordinates. Notice that the samples A and B to not necessarily fall upon particle positions. Also notice that the support radius of A encompasses 7 particles while the support radius of B only encloses 3 particles, therefore A is darker than B. This is shown in the final result, the lower part of the wave has more particles and is therefore darker.

Once the texture has been updated the raycasting shader is launched. The most important initial step is determining the rays along which to sample the texture. Krueger and Westermann propose a two-pass scheme for calculating the ray directions: first the front and back faces of a unit cube are rendered, with the red, green and blue values of each vertex determined by the x, y and z coordinates. This is depicted in figure 7.6a. By subtracting these values a direction vector is determined for use in the second pass which traces through the volume. Instead we make use of a single pass method described by Rideout [57]. As mentioned in section 5.3.2 each pixel on screen fires a ray into the scene. These position of these coordinates ranges from -1 to 1 along the x and y axes and are accessed via the built-in gl_FragCoord variable. We construct a 3D vector with the x and y values of screen coordinates and the z value as the focal length of the camera which is passed in as a uniform variable. This vector is multiplied by the modelview matrix to give a worldpspace ray direction. The begin and end points of the ray are then calculated by performing a ray-box intersection from the camera position along this directed ray (figure 7.6b).

From the start point the shader begins stepping towards the end point, sampling the



(a) (b)

Figure 7.6: Two methods of calculating the view rays along which to perform stepping. Figure 7.6a shows the method proposed by Krueger and Westermann: a cube is rendered out of 6 cubes and the vertices of these cubes are coloured according to the 3 primary colours (red green and blue). The cube is rendered into an FBO, once with depth testing and once without, and subtracting the colour value of the one from the other gives a ray direction. Figure 7.6b shows an alternate method proposed by Rideout which is faster since it does not require extra rendering passes: the screen coordinate of each pixel and the focal length of the camera are used to calculate eye-space rays which are then transformed into world-space using the modelview matrix. A ray-box intersection test is then performed to give the start and stop points for stepping.

texture at each interval and accumulating opacity according to the DVRI (equation 5.2). Stepping will terminate if either the opacity is 1 or the end point is reached. Following this procedure will produce an image similar to the x-ray optical model depicted in figure 5.7b since it does not model surface lighting, reflections or refractions. In order to produce these effects an isosurface needs to be extracted, which is done as follows. While stepping through the texture the shader checks when it first encounters a sample of opacity higher than a certain threshold, at this point the surface normal is determined by sampling one stepsize along each axis and calculating the gradient from the difference. The normal is stored for later use in lighting and reflection, but it is also used to perform refraction. The ray direction is modified by refracting it around the surface normal according to the refractive index of the fluid (which is a tunable parameter). Since the refracted ray no longer faces the original stopping point we need a new stopping criterion, which is simply to measure the distance between start and stop points up front then keep track of how far the ray has travelled. If the refractive indices are not too large we can assume that the rays will not terminate too far short but we add a multiplier to ensure that no space of the volume is skipped. Overshooting is not a problem since out of range coordinate reads on a texture are clamped, if an edge sample is non-opaque then it doesn't contribute any opacity and if it is opaque then it is usually below the surface so the artefact is not noticeable.

The fluid is rendered inside a skybox which is produced using an OpenGL cube map. The final image is composited by weighted contributions of the following elements: 1) the refracted background 2) the semi-translucent fluid volume 3) the reflection and specular off the surface of the fluid.

Because the sampling rays are all emitted from the eye position and project outwards in a cone, this method is susceptible to a particular slicing artefact whereby the texture samples align into a ring-line pattern, shown in figure 7.7. Two methods can be used to alleviate this problem: hitpoint refinement and interleaved sampling both of which enhance the isosurface extraction and are described by Scharsach [60]. Hitpoint refinement starts from the first point where the ray sample is above the threshold and iteratively seeks back and forth along the ray direction in decreasing steps to narrow down the position of the surface more precisely. Interleaved sampling jitters the start positions of the rays along the z axis before beginning stepping, thereby ensuring that their sampling patterns to not sync up. Since random numbers are difficult to generate in shaders a noise texture is used. The effects of these two techniques are discussed further in chapter 8.

We note that the well known acceleration technique of empty-space skipping was included in the original paper and can be applied to this method, but it would require rebuilding a hierarchical spatial query data structure (such as a kd-tree or octree) every frame, which would likely outweigh any performance benefits. It is also worth mentioning that 3D textures can consume relatively large amounts of memory and therefore the resolution is limited. Fraedrich et al [23] propose transforming the rectilinear grid coordinates of texture into a frustrum shape based on the viewing perspective. This results in a perspective grid which allows adaptive sampling according to viewing direction and makes better use of the limited available resolution.

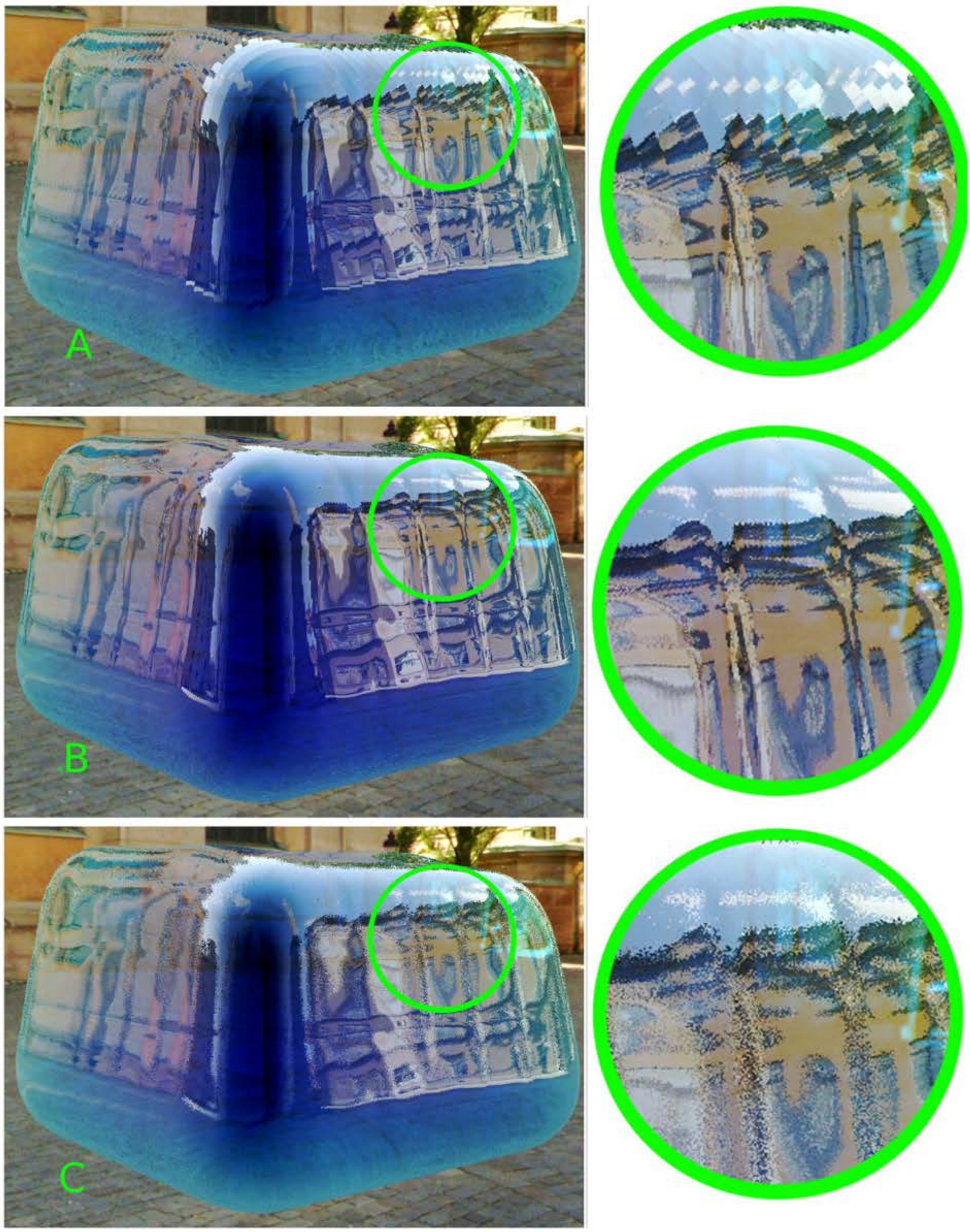
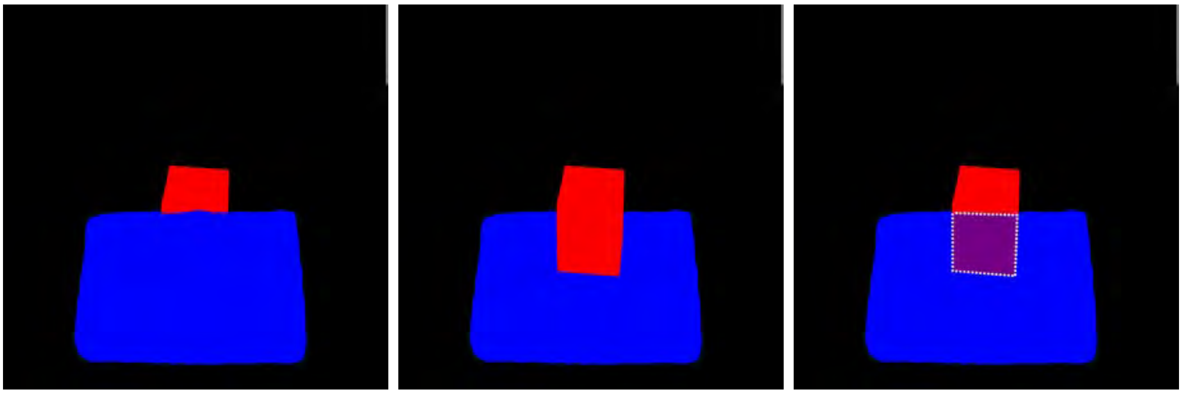


Figure 7.7: Slicing artefacts are visible in (a) due to the fact that sampling rays all start aligned at the same Z value. As they move outwards these rays sample in concentric circles which become visible in the final image. It is especially noticeable when the fluid moves since the rings remain stationary relative to the view. The hitpoint refinement method is shown in (b) which eliminates the ring artefacts (note how the clouds above the building are solid rather than staggered) but it is costly to implement. A faster option is to jitter the starting point along the Z axis, this is shown in (c) but it introduces noise into the final image.o

7.2.3 Solid Rendering

Incorporating solid rendering involves some modifications of both techniques. Visually there are two effect we need to achieve in order to display submerged solids objects: colour attenuation and refraction. Unsubmerged parts of solid objects must not have these effects. Will focus on colour attenuation first. In both cases we need to be able to distinguish between fluid and solid particles. We do this by simply storing all the solid particles after the fluid ones in the original buffer. This way the test for whether a particle is solid or fluid is to simply check whether its index is greater or less than the dividing index.

In order to render solids for the splatting method we need to add two more passes. For the depth and thickness passes we only rendered fluid particles (i.e. ones whose index is below the solid index) but now we render all of them colouring fluid ones blue and solid ones red. In the first pass we enable the depth test (shown in figure 7.8a) while in the second we disable it (figure 7.8b). Thus, any point which is red in both passes is an unsubmerged solid, while a point that is blue in the first but red in the second is a submerged solid. Submerged solids will have colour attenuation applied while unsubmerged ones will not.



(a) (b) (c)

Figure 7.8: The fluid particles are rendered in blue and the solid particles are rendered in red. In order to construct the mask for finding submerged solids, all the points first rendered with depth testing enabled (a) and then with depth testing disabled (b). Areas that are blue in a and red in b indicate solids that are below the surface, the difference between a and b is highlighted in the dotted area in c.

The raycasting method follows a similar colouring scheme, but the colour is applied while populating the 3D texture. When a texture cell is performing an SPH summation it tests the index of each surrounding point. Solid points contribute red while fluid points contribute blue. When performing raycasting the colour of each point is tested, when a red point is encountered the current opacity of that fragment is checked. If the current fragment has no opacity then an unsubmerged solid has been found, but if the current fragment already has some opacity then a submerged solid has been found and colour attenuation is applied.

In both cases the solids are rendered in yellow, since it is made up of both red and green. Water is generally blue and has little attenuation at this wavelength, but more attenuation at green and red wavelengths. Therefore yellow will show the effect of colour attenuation most clearly.

Refraction of submerged solids is difficult to achieve in the splatting solution, since submerged particles are occluded in figure 7.8a and depth information is discarded in 7.8b. Rendering solid particles only in a separate pass (with depth testing enabled)could potentially yield the required depth information, but the limited resolution of the depth buffer makes comparison of values between two different buffers unreliable. Incorporating refraction of solids into the raytracing method is simpler since the depth information is readily available, but more work is currently required on this.

Chapter 8

Results

The primary outcomes of this work have been to add solid-fluid coupling and realistic rendering to Hoetzlein's GPU based SPH simulation [29], and in this chapter we will evaluate the results. The rendering has two criteria to assess: visual quality and performance. Since our aim is to maintain interactivity we need to make trade-offs between these, and we will examine some of the parameters that can be configured in both of our rendering implementations in order to reach a good balance. Our solid coupling will be evaluated to show its ability to handle several different scenarios, including a discussion of its current limitations and areas for improvement. Finally, we will demonstrate the incorporation of solids into our rendering solution, though it is currently fairly rudimentary and can only display untextured solids. Rendering of textured surfaces from point data is beyond the scope of this work, though it has been done previously by Botsch on the GPU [15].

[1]

8.1 Fluid Rendering

We have implemented two rendering approaches, splatting and raycasting, which aim to reproduce the key visual characteristics of liquids. As described in section 5.4, these are as follows: color attenuation, which is governed by Beer's Law. Light is absorbed as it goes through a fluid, this is a function of its thickness with different wavelengths a different strengths. Reflection and refraction are governed by two laws: 1) Snell's Law, which determines the angle at which light is refracted as it passes from a medium of one optical index to another 2) The Fresnel equations, which determine what proportion of light is reflected and what proportion is refracted, based on the angle of incidence. We will not be covering caustics in this work.

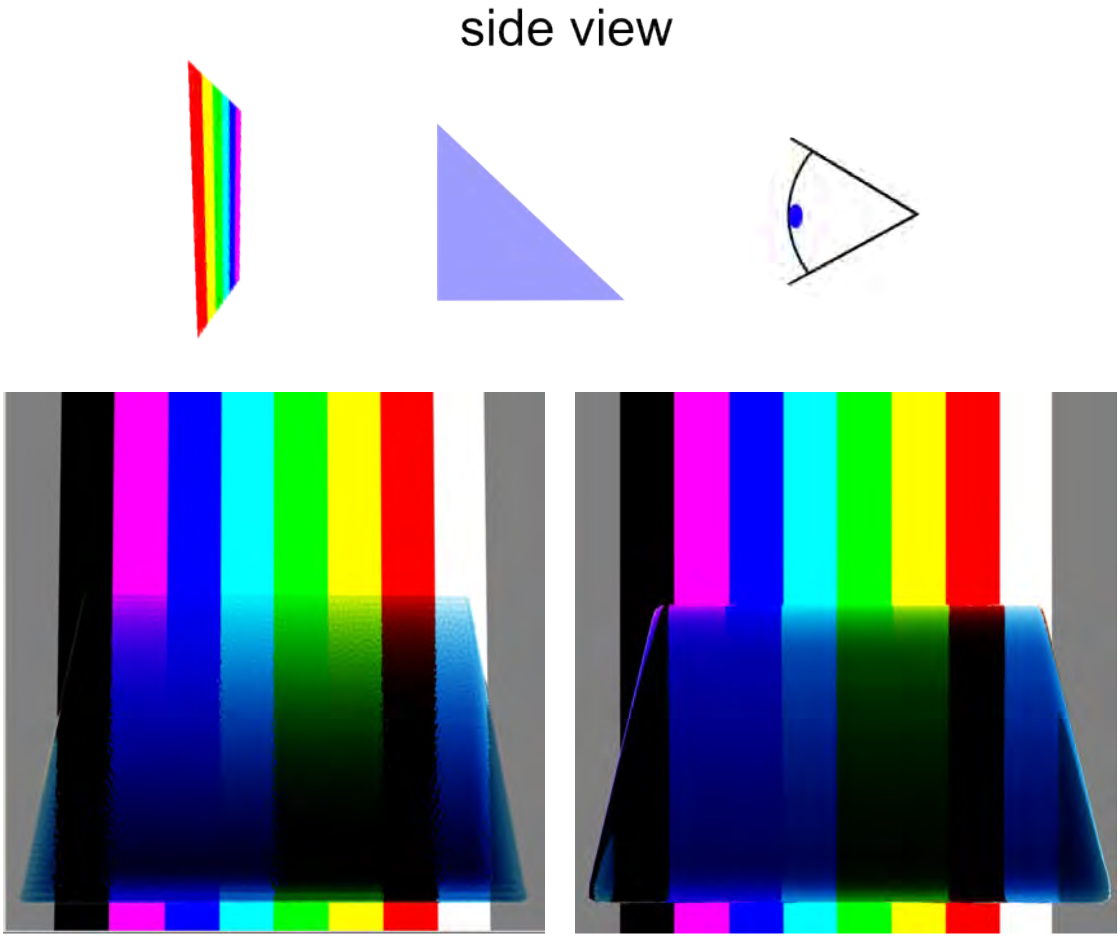
[1] All cubemaps are by Emil Persson [55].

8.1.1 Visual Quality

Our aim is for the visual output to be both as physically accurate and aesthetically pleasing as possible. To this end we will begin by presenting some simple test scenes designed to verify the correctness of the individual components. In some cases a diagram is provided to clarify the layout of the scene. All screenshots in this section are taken with parameters configured for maximum visual quality, but this has a significant performance impact and means that in some cases this the framerate is well below our target range of 24-30 fps for smooth animation. In section 8.1.2 we will discuss this further and showing some compromises that can be made in order to gain more speed, and demonstrate the artefacts these compromises might introduce.

The outputs from our test scenes are shown below. Figure 8.1 tests that light absorption occurs correctly by viewing a set of coloured panels through a volume of water. The water is wedge-shaped to make the bottom thicker thereby absorbing more light, also each colour is absorbed at a different strength which can be seen by how gradually the background goes dark. Reflection is demonstrated in figure 8.2 by rendering only the reflection part of the shaders with a ball of fluid and finally the refraction can be seen in figure 8.3 which uses a real-world image of liquid as a reference to verify that light has been bent realistically as it is transmitted through the simulated fluid.

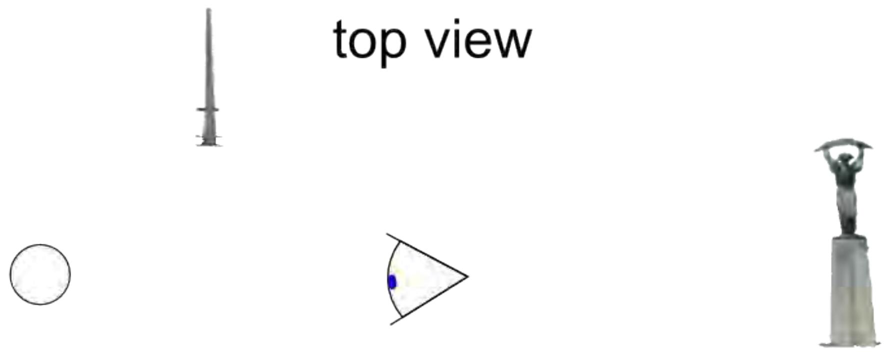
Generally, it can be seen that both methods are able to successfully reproduce the key visual aspects of liquids. The main difference between the two methods is that splatting produces a surface which is inherently slightly bumpy, while the raycasting method always produces a very smooth, glassy surface. Either look might be suitable depending on the application for which it is being used, for example: a large, fast-moving moving ocean scene will need a rough, foamy surface whereas a smaller scene like water pouring into a glass of will need a smoother surface. That said, splatting surface can be smoothed even more but that is time-consuming, a blur radius of 15 was found to give a good balance and is used in all examples below. Curvature flow, which is a better option is discussed in section 8.1.2.



(a) Splatting

(b) Raycasting

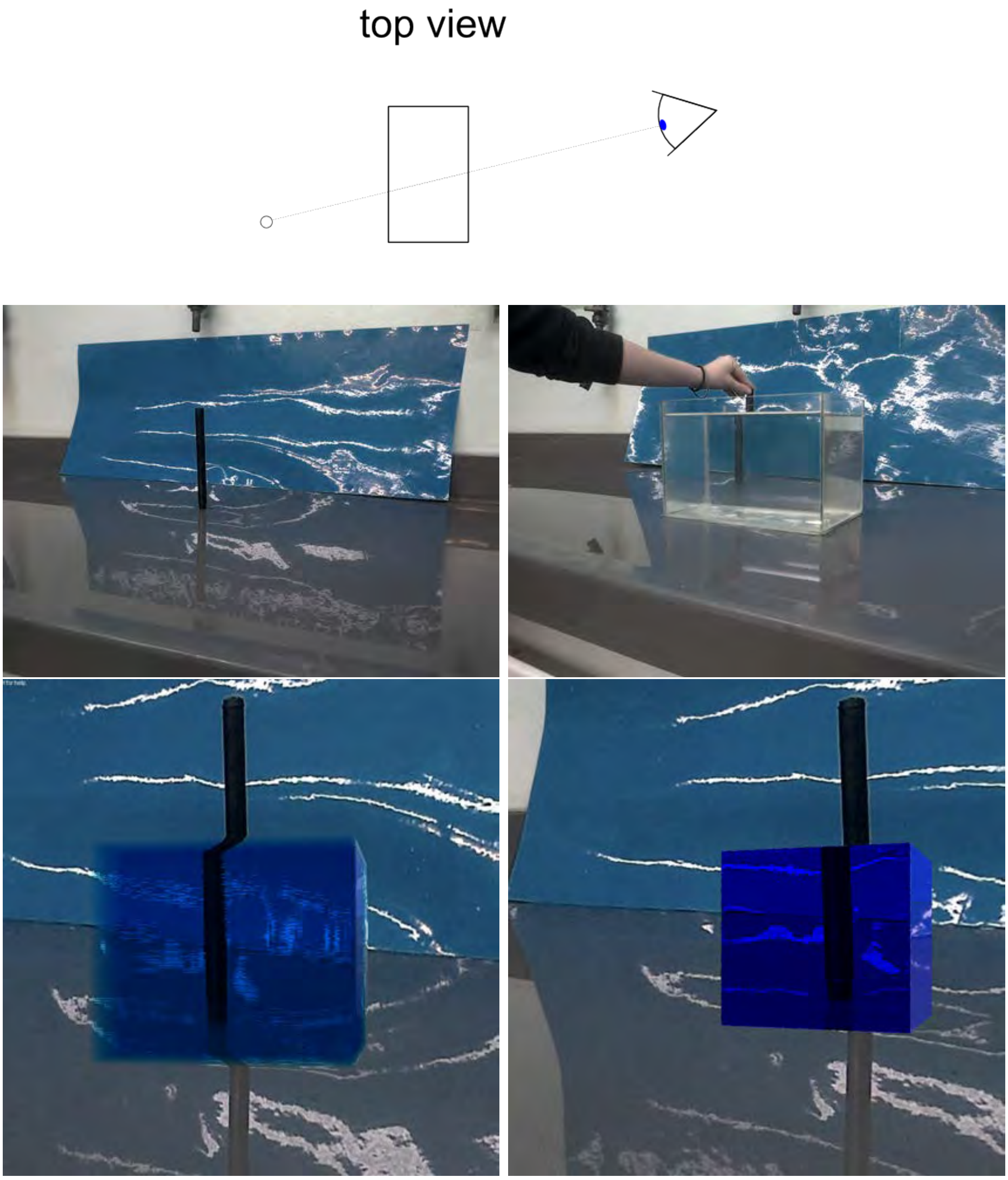
Figure 8.1: Calibration rig verifying that colour absorption due to Beer's Law is performed correctly. A wedge-shaped volume of water is placed in front of a screen containing all the primary and secondary colours. The absorption coefficients of the water are Red: 0.9, Green: 0.6, Blue: 0.1. We can see that the red column is extinguished to black most rapidly while the blue column does not darken very much at all. Green has an intermediate absorption factor, so that column demonstrates most clearly how light is absorbed gradually as the fluid becomes thicker, as do the yellow and cyan since they both contain green. The magenta column (red+blue) demonstrates how the separate channels are absorbed, all the red is gone but the blue comes through.



(a) Splatting

(b) Raycasting

Figure 8.2: A spherical ball is placed in a scene with an easily distinguishable floor and sky. These can be seen reflecting as expected on the top and bottom of the balls. Two objects in the scene are used as landmarks to verify horizontal angles: there is a pole roughly 45 degrees behind the viewer and a statue directly behind the viewer. The pole can be clearly seen in both images, while the statue is slightly distorted in the splatting.



(a) Splatting *(b) Raycasting*

Figure 8.3: The effects of refraction are more difficult to judge than for reflection, so reference photographs are used to ensure accuracy. The top row shows a black stick placed behind a tank of water, refraction causes an apparent break when it crosses the fluid boundary. In the bottom row the image of the stick (without the tank) is used as a backdrop for our two shaders, and a similar break is achieved.

Having established that these parts are working correctly we proceed to show a water-drop test in which a large drop of water falls into a still pool. The results are shown in figure 8.4 and 8.5. This test was used by Thuerey in [66] and later by Reid in [56] and Clough in [18]. It is a good rendering test because the water forms a distinctive coronet shape, which has a visually interesting and complex surface. this demonstrates each method's ability to capture fine surface detail. Compare frames 10,11,12 in both figures to see how the raytracing method produces a sharper surface. In both cases the pool contains 216,000 particles (60^3) and the drop contains 64,000 particles (40^3). An important point is that, because the raytracing method will produce a continuous surface regardless of how the underlying particles are arranged, referring back to figure 7.5 notice that even if there is only one particle in a texture cell the whole cell will be the same opacity. Coupled with built-in bilinear filtering of the texture this means that a smooth surface can be obtained with much fewer particles required than for splatting. With the splatting method the particles have to be rendered large enough to avoid gaps appearing. When rendering larger bodies of water the particles will overlap more and so can be rendered a bit larger, but to render small surface detail many more small particles are required so the cost of the simulation goes up .

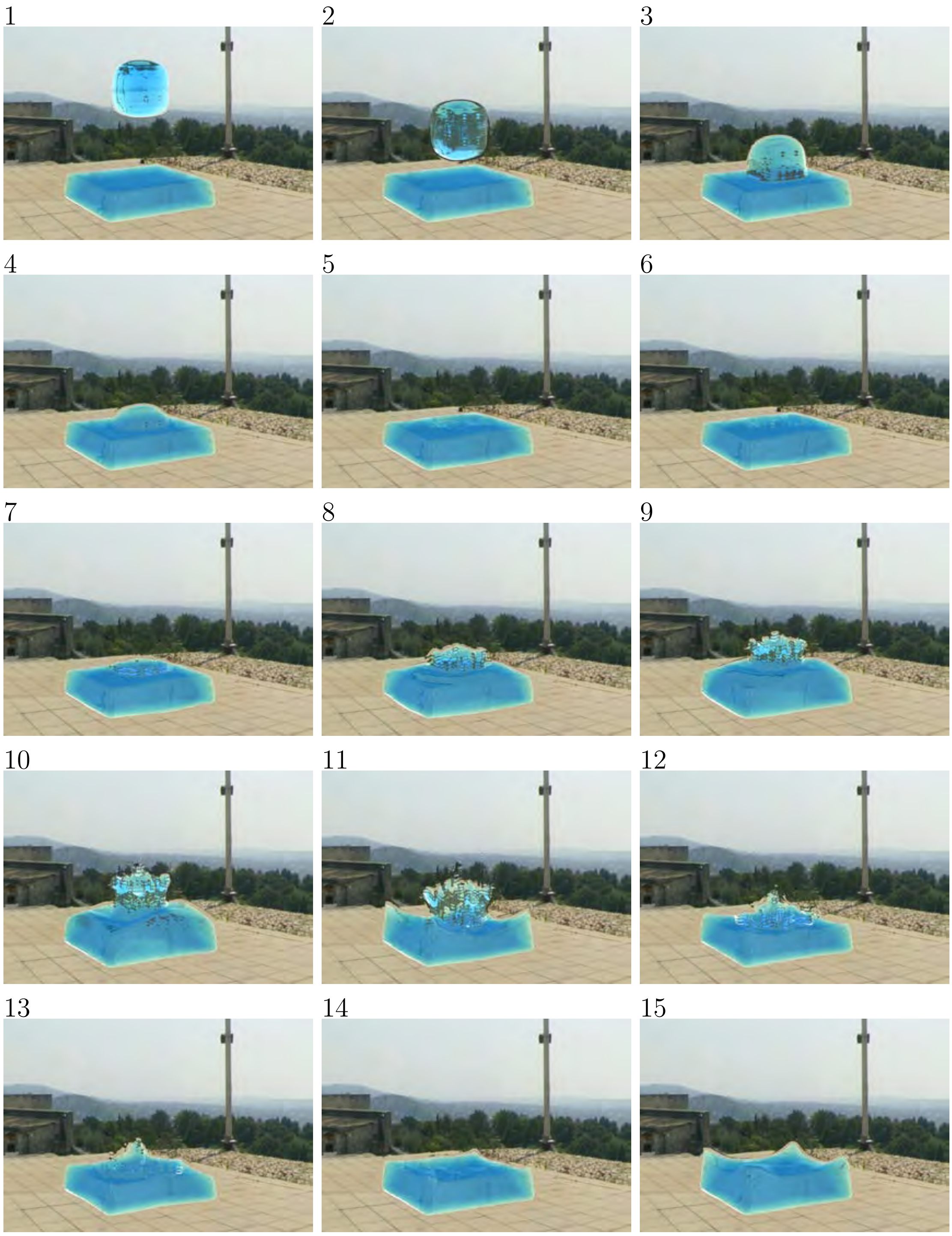


Figure 8.4: Waterdrop experiment showing a drop of 64,000 particles falling into a pool of 216,000 particles, rendered using splatting. Frame 10 shows a slightly blobby appearance since the particles are rendered relatively large to avoid gaps showing in between them. Stray particles are eliminated by thresholding the thickness value. An alternative would be to render stray particles as foam, which could be implemented in future work.

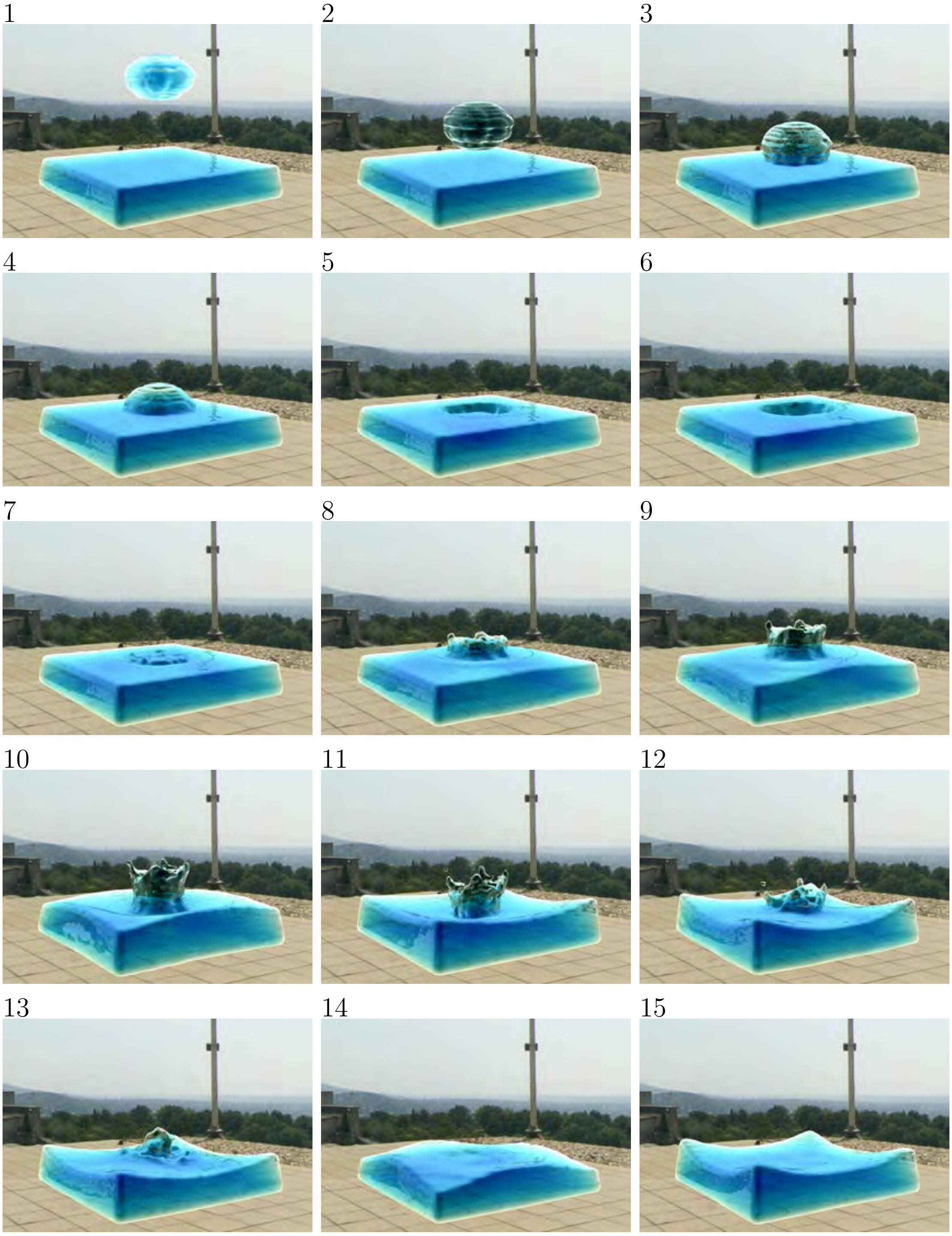


Figure 8.5: Waterdrop experiment using raycasting. This sequence was taken with a 3D texture of 256^3 in order to capture the movement of the drop and the fine detail of the splash (no visible improvement is gained by increasing the resolution further). This however meant a significant performance decrease, at this resolutions the simulation only ran at 3fps and was noticeably jerky. The results of running at a higher framerate and lower resolution can be seen in figure 8.6.

Figure 8.6: Falling drop of water with 3D grid resolution of 128^3. Tearing artefacts are visible as the fluid moves between grid cells. The higher grid resolution seen in figure 8.5 addresses this issue but at a severe performance penalty.

8.1.2 Performance

Our full system has three main phases: rebuilding the acceleration grid, running the SPH simulation and rendering. While the simulation involves lots of pairwise particle comparisons, rebuilding the acceleration grid is a linear operation takes a very small fraction of the frame time (it is less than 5% the size of the simulation phase) so we will not analyse it here. Our main focus will be on the simulation and rendering phases, and we make use of Fraps [4] to measure the framerate of our application, and we aim to maintain a rate of between 24 and 30 fps which is a generally accepted value for being considered smooth animation [68]. Note that the perceived responsiveness of the system is more complicated than just the framerate, our target range is what is required to avoid perceptible flickering in the human visual system but it does not mean that the motion of the fluid on screen is necessarily quick. If, for example, the timestep is very low then the fluid will appear to move sluggishly even at high framerates. Since the simulation is faster than the rendering, a possible technique for increasing responsiveness would be to run multiple simulation timesteps per rendered frame. We also use the system clock to measure the amount of time each phase takes within the frame itself. All tests were run on the same system, the specifications of which can be found in figure 8.1.

Our preliminary testing showed that in both methods the rendering step is significantly more costly than the simulation itself, which makes it the main bottleneck in the overall performance of our system. In order to properly measure the effect we need to begin with a baseline measurement of simulation performance with the fastest possible rendering, this being an opaque rendering of the particles, without any blending rendered in a single pass. In such a situation the limiting factor would be the number of particles. The cheapest possible way to render our particle data is using *glDrawarrays()* with GL_POINTS and the point size set to the smallest possible value. We performed an initial set of benchmarks with this configuration to find the largest scale simulation that could run smoothly within our target framerate range. An important point to note is that the performance of the simulation is affected by the shape of the fluid, for example, a flat body of water will have many particles with few neighbours at the surface. This will therefore run faster than a deep pool where most of the particles are fully surrounded by neighbours. Taking this into

CPU	Intel Core i5-3570K
CPU Clock Speed	3.4 GHz
CPU Memory	8G DDR3
GPU	NVIDIA GeForce GTX 660 Ti
GPU Generation	Kepler
GPU Clock Speed	980MHz
GPU Memory	2 GiB GDDR5
GPU Memory Bandwidth	144.2 GB/sec
GPU Cores	384
CUDA Runtime Version	4.2

Table 8.1: Hardware on which testing was performed.

account, our approach was to use progressively larger cubes while ensuring that the shape was always the same. We can see from table 8.2 that the simulation runs smoothly up to 216000 particles without rendering. Using this as our baseline, we proceed to introduce our two rendering solutions and then work backwards to smaller scales to see at what point we again reach a framerate between 24 and 30. The outcome of this procedure is shown in tables 8.3 for splatting and 8.4 for raycasting.

Dimensions	Particles	FPS
10x10x10	1,000	390
20x20x20	8,000	333
30x30x30	27,000	206
40x40x40	64,000	100
50x50x50	125,000	55
60x60x60	216,000	32
70x70x70	343,000	22
80x80x80	513,000	16
90x90x90	730,000	10
100x100x100	1,000,000	7

Table 8.2: Baseline simulation performance with cubes of increasing size and basic point rendering.

Splatting

The performance of the splatting technique is heavily dependent on the pixel area the fluid occupies on screen. This is because the shading for fluid pixels is performed several times (once for each pass), whereas the shading for the non-fluid pixels only occurs in the final pass and is a single cubemap texture fetch which is relatively fast. In all cases the system was run in a window of 1024x1024 but the camera was zoomed out such that the fluid volume occupied increasingly smaller areas of the screen. These areas were determined by progressively halving the height and width of the fluid on screen, these dimensions thus require smaller number of pixels compared to the full screen area (1024x1024) namely one quarter (512x521) and one sixteenth (256x256) of the pixels respectively. These results are shown in table 8.3, and important point in these results is that while the smallest dimensions timings are much smaller than the largest they are nowhere near 1/16th. As an example, compare the draw time of the row with 8000 particles: 19ms at 1/16th versus 35ms at fullscreen, which is 54.28% instead of the expected 6.25% This is most likely due to some constant overhead in the drawing process. Note that the timing of the simulation is constant with respect to the screen size and is therefore placed on the far left. Note also that the overall frame time includes the reconstruction of the acceleration grid as well as various housekeeping tasks. We therefore do not expect the sum of the simulation and rendering phases to precisely equal the overall frame time. One final point is that, for all particles counts and all screen sizes, the rendering takes significantly longer than the simulation.

		1024x1024			512x512			256x256		
Particles	Sim (ms)	FPS	Draw (ms)	Frame (ms)	FPS	Draw (ms)	Frame (ms)	FPS	Draw (ms)	Frame (ms)
1000					36	25	27	47	18	21
8000	2	26	35	38	36	24	27	44	19	22
27000	4	22	39	44	31	27	32	40	20	25
64000	7	18	46	54	26	30	37	34	21	29
125000	13	14	60	73	21	34	48	26	24	38
216000	22	11	67	90	16	38	61	20	27	49

Table 8.3: Framerates and timings of splatting technique for various numbers of particles and various onscreen areas occupied by fluid. The simulation time remains constant regardless of the rendering time and is therefore displayed in a separate column, while each screen area displays the rendering time and the overall frame time (including simulation and various overhead). It can be seen that in most cases the rendering occupies a large proportion of frame time.

Looking at figure 8.7 we can see that between 27000 and 64000 particles at roughly half screen size is a good feasible range for this rendering technique. We can see that the majority of the time is spent in the blurring pass applied to the depth map, the purpose of which is to reduce the inherent bumpiness of constructing a surface from point sprites. As indicated in section 7.2.1, we make use of a gaussian blur of radius 15, the cost of this can be reduced by using a smaller radius gaussian blur, but a faster method would be curvature flow [67]. We leave exploration of this technique for future work.

Following the blurring we see that the thickness pass is the second most expensive, as is to be expected, since blending requires accumulation of opacity from all points. Also all the points need to be rasterized, whereas the depth pass will only rasterize points which pass the depth test. This step is what ultimately amounts to evaluating the Volume Rendering Integral (section 5.3.1). The rendered size of the particles does have a significant effect on this pass though: larger particles give a less bumpy appearance but overlap more and thus require more rasterization and blending. The particle size can thus be adjusted to balance performance and aesthetic requirements.

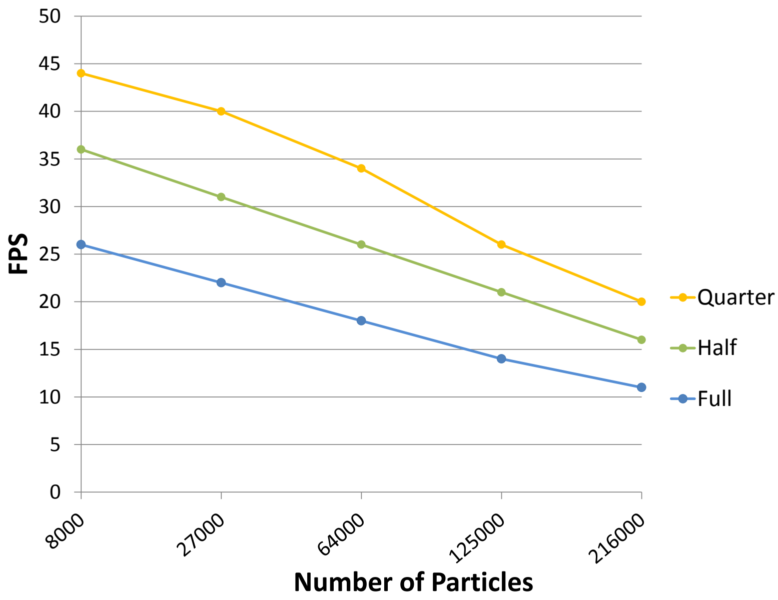


Figure 8.7: Performance comparison of cases where fluid occupies different amounts of onscreen area. Half the screen (512x512) provides a reasonable range of performance values.

Raycasting

Unlike with splatting, the performance of raycasting is much less affected by the number of particles. In table 8.3 we can see that the framerate for 1000 particles is always more than double the framerate of 216,000 particles (for exmaple 36fps vs 16fps at 512x512). Meanwhile in table 8.4 the increase is only 60% (20fps vs 12 fps). The reason for this is that this method spends most of its time performing raycasting through the 3D texture, which is an operation that is independent of the number of particles. We can see, by breaking down the drawing phase, that this time remains constant while the time taken to populate the texture increases with the number of particles. Overall in table 8.4 the time taken for populating the texture grows in a rate very similar to the time taken to run the simulation. The reason they are not identical is that the simulation kernel samples density by performing an SPH summation at the position of every particle, whereas the texture population kernel samples in a uniform 3D grid corresponding to texture cells. This means that small variations are to be expected when particles do not fall exactly on grid boundaries.

The size of the 3D texture in table 8.5 has a more significant effect on performance, since a larger texture requires more threads to populate it. Texture resolution is the biggest determinant in final visual quality: too low a texture resolution will result in blockiness of the fluid volume. We have found that 128^3 is enough to handle most situations but, as mentioned in section 8.1.1, higher resolution is required to capture small volumes of faster-moving fluid. We also note that all texture dimensions listed here are cubic,

Particles	FPS	Sim	Draw		Frame
			Populate	Raycast	
		(ms)	(ms)	(ms)	(ms)
1000	20	2	5	42	50
8000	19	2	7	42	52
27000	18	4	9	42	56
64000	16	7	10	42	60
125000	14	13	13	42	69
216000	12	22	15	42	80

Table 8.4: Performance of volume raycasting method for a 128^3 *texture and various numbers of particles. The time taken to perform particle simulation is low relative to the overall frame time, and the time taken to perform raycasting is constant with respect to the number of particles. The time taken to populate the 3D texture varies with the number of particles, which is because this operation performs a neighbour search algorithm similar to an SPH kernel.*

i.e. have equal dimensions along all sides. While rectangular textures are supported by OpenGL they lead to stretching of the final image.

Texture Dimensions	FPS	Texture Populate (ms)
32^3	18	2
64^3	17	5
128^3	14	20
256^3	6	92

Table 8.5: Rendering performance of texture population for 64000 particles at various texture dimensions.

Finally, we tested the CUDA kernel that populates the 3D texture by setting various block dimensions and found that larger blocks execute faster, but with diminishing returns. The reason for this is that the kernel is entirely IO bound, there is no register or shared memory usage limiting the number of threads. Therefore once several large blocks are running they will cover each other's memory fetch latency and there is no reason to run more smaller blocks.

Block Dimension	FPS	Texture Populate (ms)
2^3	6	110
4^3	13	64
8^3	14	62

Table 8.6: Rendering performance of texture population for a texture of 128^3 *with various CUDA block sizes.*

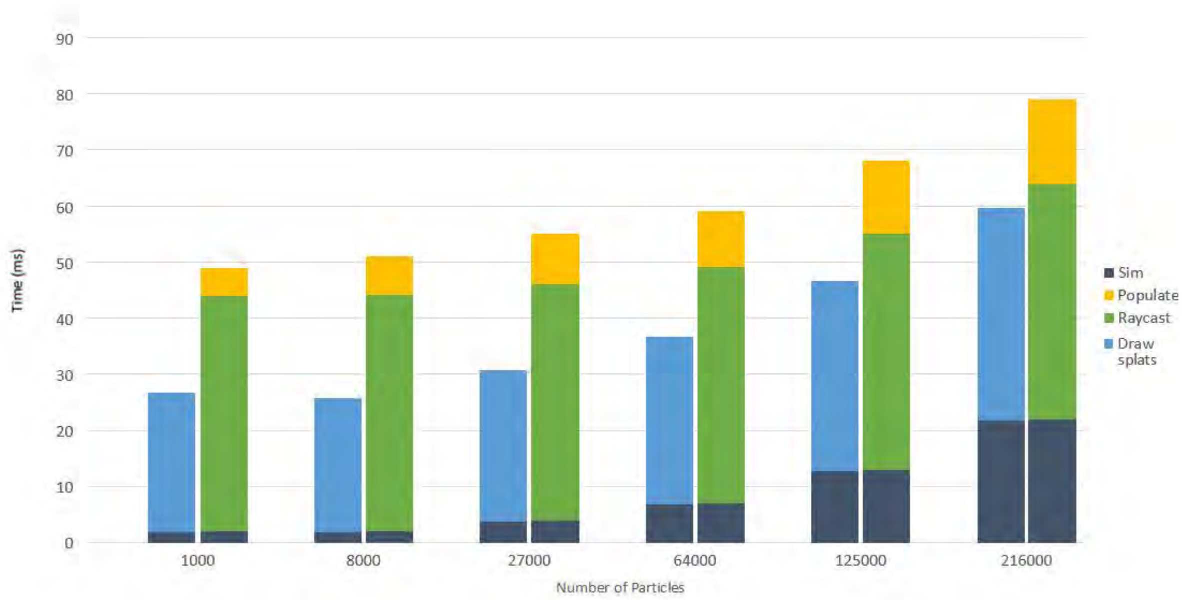


Figure 8.8: Comparison of splatting and raycasting rendering methods with reasonable defaults selected based on tables 8.3 and 8.4. For splatting, the onscreen area is 512x512 while for raycasting the texture volume is 128^3. We can see that the rendering dominates the frametime. We can also see that for raycasting the texture populating step is relatively quick for fewer particles but begins to take comparatively more frame time as the number of particles increases. Although it is not illustrated it should be born in mind that splatting is very sensitive to zooming in and out, so the framerates depicted here will fluctuate, while raycasting takes a constant amount of frametime regardless of zoom.

Comparison

There are many different parameters which effect the performance of the splatting and raycasting methods, making direct comparison difficult. For splatting the screen resolution is a key parameter and based on 8.3 we have selected 512x512 since it provides a range of framerates in our desired range. For raycasting we have selected a 3D texture size of 128^3 since this provides sufficient detail without excessive cost to populate. With these two parameters locked we can compare the two methods based on the number of particles in the scene. This comparison is shown in 8.8. We can see that for both methods the majority of the frame time is taken up by the rending, with simulation being relatively small. This further emphasizes that rendering is the primary performance bottleneck for interactive fluid simulations. Overall we can see that with these parameters the splatting performs better, but bear in mind that the framerate is very sensitive to zooming in and out while the raycasting provides a stable framerate regardless of how much area the fluid occupies on screen.

8.1.3 Memory Consumption

Graphics cards have limited memory available, so memory consumed by each technique is an important consideration. Tables 8.7 and 8.8 show the memory consumption for each method. In the case of raytracing, memory is consumed by the 3D texture, but at 128^3 we can see that the memory usage is fairly moderate (32 megabytes) and with acceptable visual results in most cases. A resolution of 256^3 consumes significantly more memory but is able to capture faster moving fluids without tearing (as described in figure 8.6).

Texture Dimensions	Memory (MB)
64^3	4
128^3	32
256^3	256
512^3	2048

Table 8.7: Memory consumption for various 3D texture dimensions.

Memory consumption for splatting is actually higher since several passes are required, as can be seen in table 8.8: 80MB of memory are required to render at 1024x1024, whereas an equivalent result can be achieved using only 32MB of 3D texture memory for raycasting at 128^3. If higher resolution is required for raycasting, then again splatting wins. Memory consumption can be reduced when splatting by rendering some passes at lower resolution (especially in the case of the thickness pass). This would also increase performance and is an avenue worth exploring in future work. Rendering and upsmapling could also have the benefit of using the texture interpolation hardware to implicitly apply smoothing in place of the blurring pass which would further improve performance.

Texture Dimensions	1 pass (MB)	5 passes (MB)	7 passes (with solids) (MB)
256×256	1	5	7
512×512	4	20	28
1024×1024	16	80	112

Table 8.8: Memory consumption for splatting with various 2D texture dimensions. The memory usage of a single texture in an offscreen FBO is given in the first column, but since our method uses 5 passes the overall memory consumption is given in the second column. Incorporating solid rendering requires 2 extra passes which is displayed in the third column.

8.2 Solid Rendering

The results of the solid rendering for both techniques can be seen in figure 8.9, in both cases we display a solid box fully unsubmerged, partially submerged and then fully submerged. In terms of performance, the splatting requires two extra rendering passes which consumes some memory storing the FBOs. The points in these passes are rendered smaller than in the thickness and depth passes and no blending is performed, so they are significantly faster than the other passes.

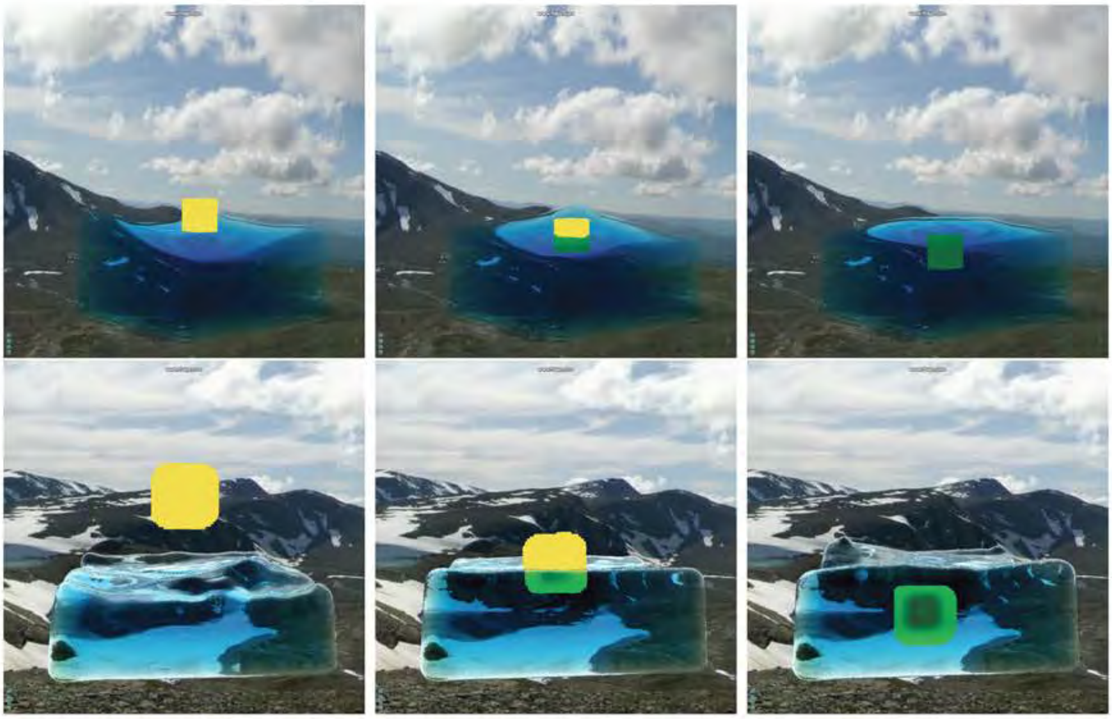


Figure 8.9: Solid box rendered using splatting method (top) and raycasting method (bottom).

8.3 Solid-Fluid Coupling

As with the rendering, we begin by verifying that a simple case works as expected and then move on to more complex demonstrations. Our system only models only hydrodynamic forces, as described in section 2.4 these are the forces which push objects and fluids, as opposed to viscous forces that exert drag. We use the scenario of a boat floating in a wave tank (depicted in figure 8.10) to test several properties of solid-fluid interaction. Reading from top to bottom, the right and left columns show boats of different masses interacting with a wave moving from left to right. This scene is rendered showing the particles as opaque balls in order to more clearly see that the interior of the boat is hollow. Intuitively, the expected behaviour as a wave passes a boat would be for the boat to rise front-first as the water pushes underneath it and then settle back down once the wave has passed. At the same time the water should break around the bow without penetrating through. We can see these expected behaviours occurring in both columns. The difference in mass between the two boats has an effect on how they behave, specifically the lighter boat has more buoyancy and is pushed slightly higher out of the water as the wave goes under it, while the heavier one sits lower in the water.

Moving on from a single floating object, figure 8.11 shows a set of boxes stacked in a corridor and then submerged by water washing past. This scene is inspired by a flood scene from the movie "*Day After Tomorrow*" [3] and demonstrates two important features: handling multiple solid objects and solid-solid interaction. As noted in section 2.4, handling contact between particle-based rigid bodies is a difficult problem, so we address this issue by wrapping our solid objects with Bullet collision shapes. This allows the boxes to be stacked stably at the beginning of the simulation and to touch each other with appropriate frictional forces. Invisible walls, defined by Bullet constrain the boxes to not fly out of the corridor when pushed by the water.

Table 8.9 shows the performance characteristics of the solid elements. The same corridor scene as above was extended by gradually adding more columns of boxes. The number of fluid particles was held constant at 27,000 which is close to the initial number of solid particles. It can be seen in the first row that when the number of solid particles is the same as the number of fluid particles, the time spent performing solid interaction calculations are very small compared to the overall frame time. The separate phases involved are: 1) transferring forces from the GPU to the CPU 2) accumulating the rigid forces applied to each body 3) physics integration (using Bullet) 4) transforming solid particles to their updated positions based on the movement of their parent bodies. We have collapsed steps 2 and 4 under the heading "CPU Calculations" since they both increase linearly in the number of particles, while the Bullet Physics stage only increases in the number of bodies. The "Solid Total" column is the sum of the force transfer, all CPU calculations and Bullet Physics. We can see that overall the solid-fluid coupling introduces very little overhead into the system. One point worth noting is that as the number of solid particles grows much larger than the number of fluid particles, most solid particles are no longer touching the water. This means that fewer force and density calculations are being performed. This is why the overall fluid simulation time does not grow very much. In any situation where solids are floating on the water we can safely expect there to be many more fluid particles than solid, and we can therefore be confident that the transfer and CPU-based calculations are not a bottleneck.

Num Boxes	Box Particles	GPU/CPU Transfer (ms)	Bullet Physics (ms)	Particle Position (ms)	Solid Total (ms)	Sim Total (ms)	Solid/ Sim (ratio)
15	30,000	0.08	0.2	1.6	1.8	31	0.06
45	90,000	0.2	0.5	4.9	5.7	70	0.08
90	180,000	0.4	1.0	10.0	11.4	76	0.15
180	360,000	1.0	1.9	20.1	23.0	80	0.28
360	720,000	2.0	3.8	40.0	45.8	104	0.44

Table 8.9: Boxes of 2000 particles each are added to the corridor scene while the number of fluid particles is kept constant at 27,000. The first 3 columns describe the different phases of solid interaction calculations: solid particle forces and positions are transferred between GPU and CPU, Bullet physics updated rigid body positions based on incoming forces and solid particle positions are updated relative to their parent bodies. It can be seen that the position updates take by far the largest amount of time since they involve several CPU multiplications and additions per point. Finally the last 3 columns show that the time taken to process solid objects is small compared to the overall simulation time unless there are far more solid particles than fluid ones.

Finally, one of the important strengths of defining solid objects in terms of particles is that boundaries can be arbitrarily shaped. Figure 8.12 shows fluid along the inside of a curved, tubular surface. This type of situation may arise in scenes where fluid needs to interact with environments such as tunnels, rivers or terrain with static obstacles.

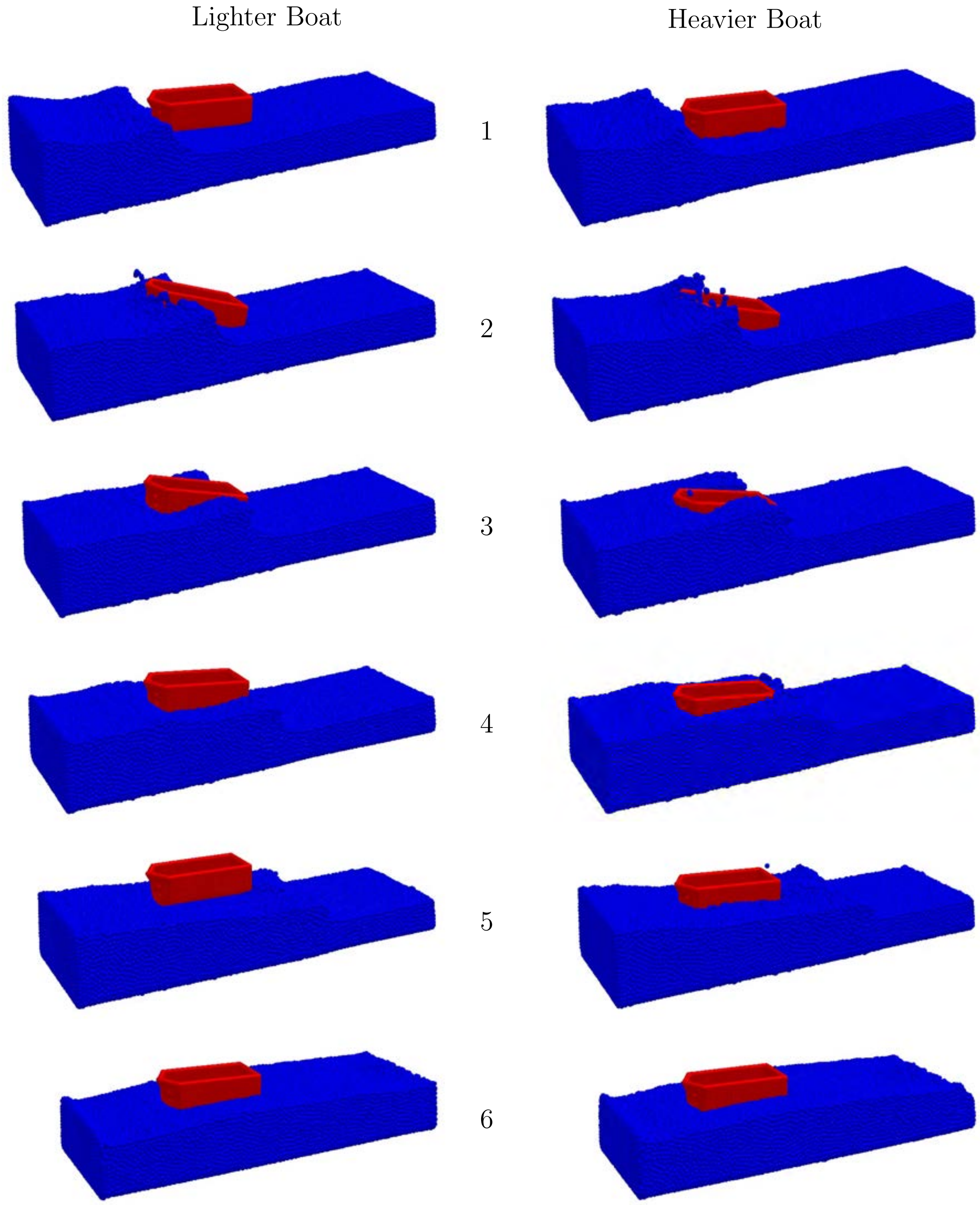


Figure 8.10: Two boats floating in a wave tank, the sequences read from top to bottom with the left column showing a lighter boat and the right column a heavier one. As expected, the boat rides up the wave front first and the water breaks on either side of the bow. The lighter boat is pushed slightly higher out of the water, while the heavier one stays low because it has less buoyancy, this can be seen most clearly in frames 3 and 4.

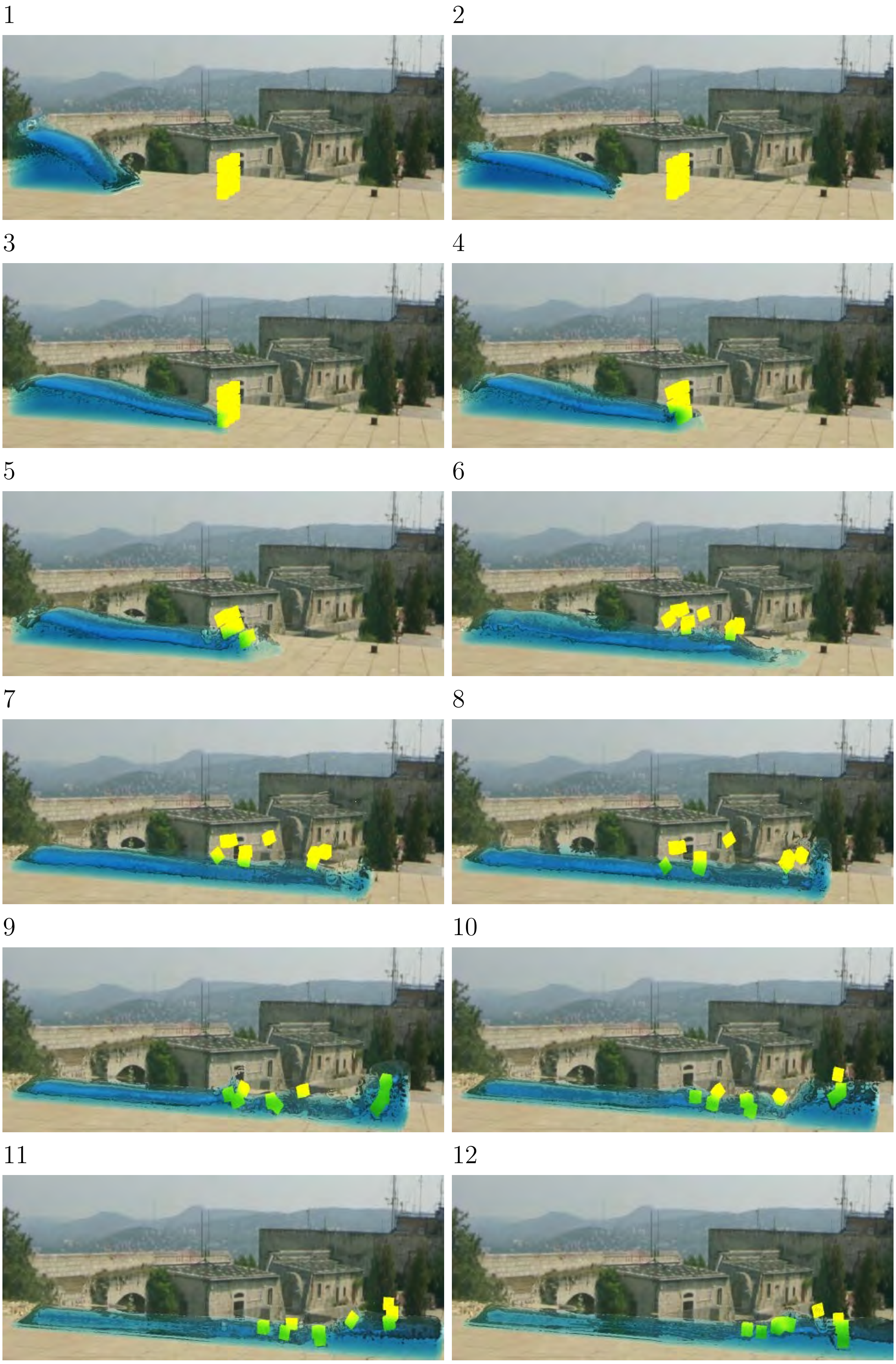


Figure 8.11: A column of water flows down a corridor, collapsing a pile of boxes and carrying them along in the flow. The scene contains 60k fluid particles and 17k solid particles and runs at 8fps.

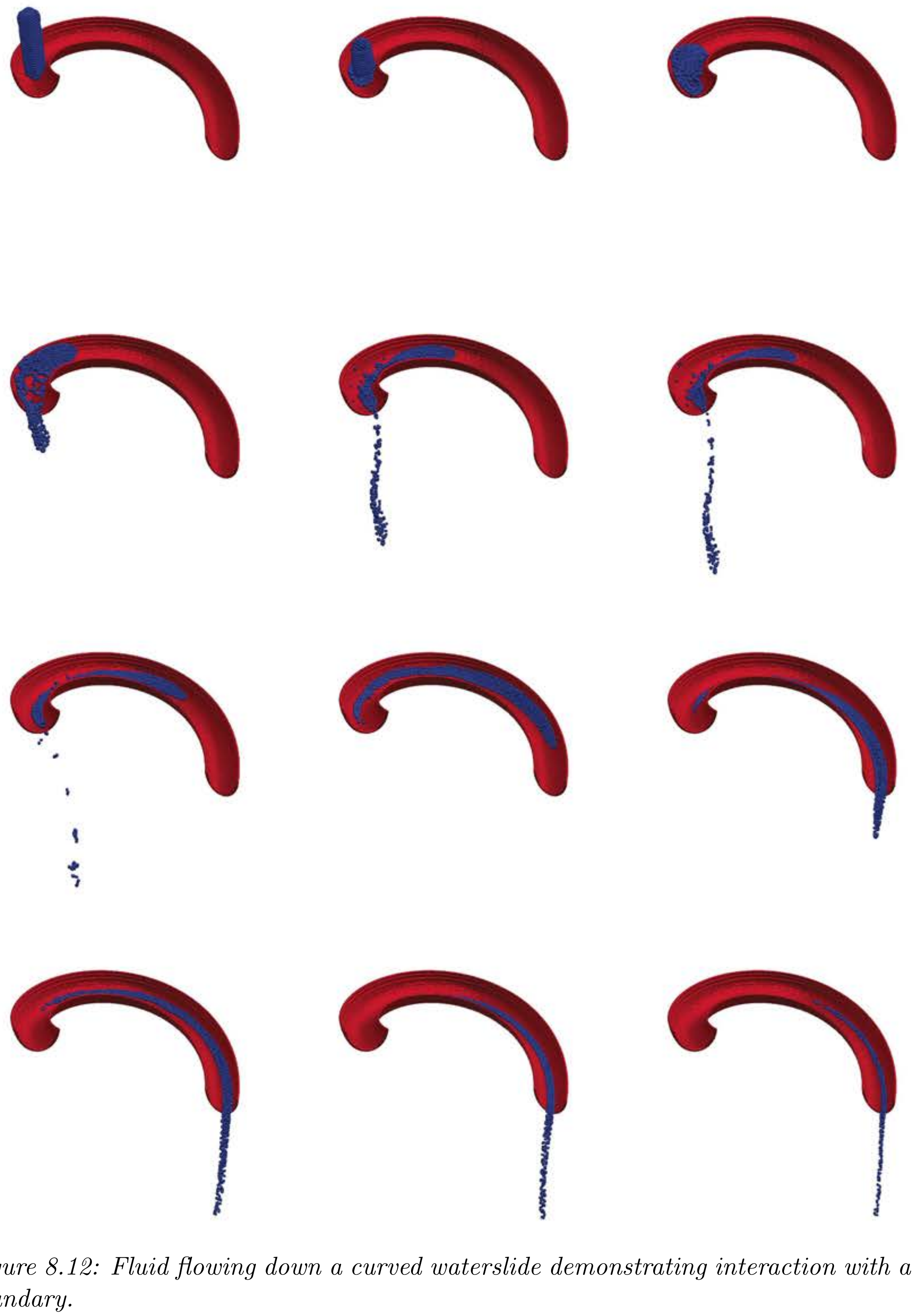


Figure 8.12: Fluid flowing down a curved waterslide demonstrating interaction with a fixed solid boundary.

8.4 Improving Interactivity

The aim of this work is to produce an interactive simulation and so far we have focused on performance purely in terms of framerate. While a framerate of 24 to 30 fps may be sufficient to maintain the illusion of continuous motion (as opposed to a slideshow effect), this measure does not say anything about the final user experience. A missing factor is how quickly the system responds to user input: let's say a user is given the ability to manipulate the liquid, if they were to rapidly tilt the container they would expect to see the liquid immediately slosh to one side. If there is a perceptible delay in this occurring then the illusion of direct control will be broken, this is referred to as *latency*.

Obviously increasing the performance of any part of the system will reduce the time it takes to generate frames and as a result decrease latency as a side effect. Nevertheless there are also some strategies which can be applied to decrease latency and thereby increase the responsiveness. These measures are largely heuristic in nature and evaluation is based on subjective aesthetic perception.

8.4.1 Scale Adjustment

The number of particles in a simulation has the most direct impact on performance, but the real-world size of the scene being represented by those particles is not explicitly defined by the simulation. Consider the corridor scene in figure 8.11 which consists of 60,000 fluid particles, based on the background this scene appears to be roughly the size of a swimming pool but the same number of particles could also represent a large dam. The main considerations would be the level of surface detail required more particles will be needed if there is going to be lots of fine splashing.

8.4.2 Timestep Size

The size of the timestep used in the simulation can be increased to produce faster fluid motion. This can lead to simulation instability, but methods such as PCISPH will be able to correct this and allow faster fluid motion.

8.4.3 Subframe Stepping

As noted in section 8.1.2, the rendering takes a lot more time than the simulation. Therefore it may be beneficial to render several substeps of the simulation for each frame. The time of all of these together should not exceed the length of the frame rendering time otherwise the result will start to appear jerky.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

As described in chapter 1, our aim was to produce a system that could interactively simulate fluid and fulfil three key criteria: 1) produce motion at a reasonably smooth framerate between 24 and 30 fps, 2) reproduce the visual characteristics of water and 3) interact with free floating rigid bodies.

We began with an existing system that could simulate up to 216k particles within our acceptable framerate range. We then added rendering and solid interaction and evaluated the performance impact of both these features. We measured the overall framerate performance, and since memory on a graphics card is a limited resource we also evaluated memory consumption.

Rendering proved to be the dominant performance factor, taking up the majority of frame time for both our splatting and raycasting implementations. For each we measured the number of particles which could be rendered at our target framerate. These measurements were taken with certain other parameters configured for best performance at reasonable visual quality. Splatting reduced the feasible particle count to between 27k and 64k while raycasting was unable to achieve the desired framerate even for 1000 particles. In the case of splatting, the performance varied drastically depending on resolution and the screen area occupied by fluid, while raycasting provided a more stable framerate at any resolution and zoom level. The performance of the raycasting is heavily dependant on the resolution of the underlying 3D texture, and high resolutions are required to capture faster moving detail without tearing. Lower texture resolutions are however adequate for slower moving fluid, and consume far less memory than the multiple passes of splatting. The major performance bottleneck of splatting is the blurring pass, which we believe will improve significantly if curvature flow is implemented instead. We also expect this to improve the visual quality.

Thus the tradeoffs between the two rendering methods are as follows: raycasting provides better visual quality with lower but more stable performance. Splatting performs generally better, but fluctuates severely as the camera moves in and out. The memory consumption of splatting is fixed since the framebuffers storing each pass consume a certain amount space, while the memory consumption of raycasting can be adjusted to be much lower at the risk of introducing artefacts.

Analysis of our solid-fluid coupling is much simpler than the rendering. Overall we found that, apart from the time taken to process the extra particles, the solid-fluid coupling introduced very little overhead. We can therefore conclude that our approach of transferring particle forces back to the CPU to leverage Bullet's integrator is a viable approach performance-wise. Although our implementation was fairly simple and did not incorporate sophisticated techniques to avoid particle penetration we found that it handled our test scenarios reasonably well without any visible artefacts occurring.

Finally, our solid rendering does not introduce any performance overhead in the raycasting, but adds an extra two passes with splatting. Also the refraction of submerged solids is easily accommodated by the raytracing method, but remains challenging with splatting.

9.2 Future Work

This thesis has touched upon all of the fundamental aspects of implementing an interactive fluid simulation, but there are many areas for improvement and advanced features which we have not implemented. This section will discuss some of these, broken down according to the different areas they relate to.

The simulation implementation currently uses a fixed spatial grid for neighbour search acceleration, but this has two drawbacks: potentially unused grid memory in areas without fluid, and also requiring that the size and shape of the fluid domain be specified beforehand which makes integration with other scenes slightly less flexible. An improvement would be to implement a spatial has grid as described by Teschner et al [49], which we also expect would not incur any performance since lookups will also be O(1). The performance of the simulation could also potentially be improved by implementing PCISPH [62] which would allow larger timesteps. The gain is less clear here since there is slight overhead incurred in performing the correction step which should result in a lower framerate, but faster perceived motion. Many references cite PCISPH usage in offline simulation where the additional cost per frame is offset by requiring fewer steps. For interactive applications larger timesteps may result in animations appearing jerky as described in section 8.4 if framerates drop too low. Finally the overall realism of simulation could be improved by implementing some of the viscosity effects described in section 2.4.

A performance improvement which would benefit the splatting rendering would be to implement curvature flow, thereby alleviating the cost associated with blurring and also avoiding excessive smoothness and potentially improving visual quality. The current implementation of the raycasting maps the entire fluid acceleration grid into the 3D texture regardless of where the fluid is actually located. A solution to this would be to implement a grid partitioning scheme as described by Fraedrich [23] which would allow empty regions to be skipped and also allow much larger fluid domains to be rendered than would otherwise fit in GPU memory. Note that this scheme refers to mapping the 3D texture to a region in space and is therefore distinct from the spatial has mentioned above. In principal the two could be used together. Apart from performance improvements, our rendering could also be improved by incorporating caustics and foam rendering.

Our solid-fluid coupling does not currently implement any technique to avoid particle penetration, the predictor-corrector scheme of Becker [14] would probably be the most straightforward to incorporate. Handling deformable solid objects would require a significant departure from our current implementation approach since Bullet could no longer be used for rigid motion integration, and forces calculations between solid particles would need to be implemented.

Finally our solid rendering in the splatting implementation can be extended to handle textured objects using techniques similar to those shown by Botsch [16].

Bibliography

[1] https://en.wikipedia.org/wiki/Marching_cubes.

[2] Bullet physics. http://bulletphysics.org/.

[3] The Day After Tomorrow. http://www.imdb.com/title/tt0319262/.

[4] Fraps. http://www.fraps.com/.

[5] Realflow. http://www.realflow.com/.

[6] Renderman. http://renderman.pixar.com/.

[7] SGI Crimson. http://en.wikipedia.org/wiki/SGI_Crimson.

[8] Adams, B., Pauly, M., Keiser, R., and Guibas, L. J. Adaptively sampled particle fluids. *ACM Trans. Graph. 26*, 3 (July 2007).

[9] Akenine-Möller, T., Haines, E., and Hoffman, N. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[10] Akinci, N., Cornelis, J., Akinci, G., and Teschner, M. Coupling elastic solids with smoothed particle hydrodynamics fluids. *Computer Animation and Virtual Worlds 24*, 3-4 (2013), 195–203.

[11] Akinci, N., Ihmsen, M., Akinci, G., Solenthaler, B., and Teschner, M. Versatile rigid-fluid coupling for incompressible SPH. *ACM Transactions on Graphics 31*, 4 (July 2012), 1–8.

[12] Batty, C., Bertails, F., and Bridson, R. A fast variational framework for accurate solid-fluid coupling. *ACM Transactions on Graphics (TOG) 26*, 3 (2007), 100.

[13] Becker, M., and Teschner, M. Weakly compressible sph for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), SCA '07, Eurographics Association, pp. 209–217.

[14] Becker, M., Tessendorf, H., and Teschner, M. Direct forcing for Lagrangian rigid-fluid coupling. *IEEE transactions on visualization and computer graphics 15*, 3 (2009), 493–503.

[15] Botsch, M., Hornung, a., Zwicker, M., and Kobbelt, L. High-quality surface splatting on today's GPUs. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* (2005), 17–141.

[16] Botsch, M., Hornung, A., Zwicker, M., and Kobbelt, L. High-quality surface splatting on today's gpus. In *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2005), SPBG'05, Eurographics Association, pp. 17–24.

[17] Bourke, P. Polygonising a scalar field. http://paulbourke.net/geometry/polygonise/.

[18] Clough, D. Lattice boltzmann liquid simulations on graphics hardware. Master's thesis, University of Cape Town, 2014.

[19] Desbrun, M., and Gascuel, M. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96* (1996).

[20] Desbrun, M., and Gascuel, M.-P. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96* (New York, NY, USA, 1996), Springer-Verlag New York, Inc., pp. 61–76.

[21] Foster, N., and Fedkiw, R. Practical animation of liquids. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 23–30.

[22] Foster, N., and Metaxas, D. Realistic animation of liquids. *Graph. Models Image Process. 58*, 5 (Sept. 1996), 471–483.

[23] Fraedrich, R., Auer, S., and Westermann, R. Efficient high-quality volume rendering of SPH data. *IEEE transactions on visualization and computer graphics 16*, 6 (2010), 1533–40.

[24] Gingold, R. A., and Monaghan, J. J. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society 181*, 3 (1977), 375–389.

[25] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2010), SCA '10, Eurographics Association, pp. 55–64.

[26] Green, S. Cuda particles. *nVidia Whitepaper 2*, 3.2 (2008), 1.

[27] Harada, T., Koshizuka, S., and Kawaguchi, Y. Sliced data structure for particle-based simulations on GPUs. *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07* (2007), 55.

[28] HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International* (2007), SBC Petropolis, pp. 63–70.

[29] HOETZLEIN, R. C. Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. In *GPU Technology Conference* (2014).

[30] IHMSEN, M., CORNELIS, J., SOLENTHALER, B., HORVATH, C., AND TESCHNER, M. Implicit incompressible sph. *IEEE Transactions on Visualization and Computer Graphics 20*, 3 (Mar. 2014), 426–435.

[31] IHMSEN, M., ORTHMANN, J., SOLENTHALER, B., KOLB, A., AND TESCHNER, M. SPH Fluids in Computer Graphics. In *Eurographics 2014 - State of the Art Reports* (2014), S. Lefebvre and M. Spagnuolo, Eds., The Eurographics Association.

[32] INA, R. T., NOBREGA, T. H. C., CARVALHO, D. D. B., AND VON WANGENHEIM, A. Interactive Simulation and Visualization of Fluids with Surface Raycasting. In *2010 23rd SIBGRAPI Conference on Graphics, Patterns and Images* (Aug. 2010), IEEE, pp. 142–148.

[33] INTEL. Core i7 specifications. http://www.intel.co.za/content/www/za/en/processors/core/core-i7-processor.html.

[34] KELAGER, M. Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics. Master's thesis, University of Copenhagen, 2006.

[35] KRÜGER, J., AND WESTERMANN, R. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003* (2003).

[36] LEIMKUHLER, B. J., REICH, S., AND SKEEL, R. D. Integration methods for molecular dynamics. In *Mathematical Approaches to biomolecular structure and dynamics*. Springer, 1996, pp. 161–185.

[37] LORENSEN, W. E., AND CLINE, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics 21*, 4 (Aug. 1987), 163–169.

[38] LOSASSO, F., TALTON, J., KWATRA, N., AND FEDKIW, R. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics 14*, 4 (July 2008), 797–804.

[39] MACKLIN, M., AND MÜLLER, M. Position based fluids. *ACM Trans. Graph. 32*, 4 (July 2013), 104:1–104:12.

[40] MEISS NER, M., HUANG, J., BARTZ, D., MUELLER, K., AND CRAWFIS, R. A practical evaluation of popular volume rendering algorithms. *Proceedings of the 2000 IEEE symposium on Volume visualization - VVS '00* (2000), 81–90.

[41] MÜLLER, M., CHARYPAR, D., AND GROSS, M. Particle- based fluid simulation for interactive applications. In *In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on computer animation, SCA '03,* (San Diego, California. Aire-la-Ville, Switzerland, 2003), Eurographics Association, pp. 154–159.

[42] Monaghan, J. J. Simulating Free Surface Flows with SPH. *Journal of Computational Physics 110*, 2 (1994), 399–406.

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

[44] Monaghan, J. J. Smoothed particle hydrodynamics. *Reports on Progress in Physics 68*, 8 (Aug. 2005), 1703–1759.

[45] Müller, M., Charypar, D., and Gross, M. Particle-Based Fluid Simulation for Interactive Applications. 154–160.

[46] Müller, M., Schirm, S., and Duthaler, S. Screen Space Meshes. In *SCA '07 Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007), D. Metaxas and J. Popovic, Eds., vol. 1, Eurographics Association Aire-la-Ville, Switzerland, Switzerland, pp. 9–15.

[47] Müller, M., Schirm, S., Teschner, M., Heidelberger, B., and Gross, M. Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds 15*, 3-4 (2004), 159–171.

[48] Müller, M., Schirm, S., Teschner, M., Heidelberger, B., and Gross, M. Interaction of fluids with deformable solids: Research articles. *Comput. Animat. Virtual Worlds 15*, 3-4 (July 2004), 159–171.

[49] Müller, M. T. B. H. M., Pomeranets, D., and Gross, M. Optimized spatial hashing for collision detection of deformable objects. Tech. rep., Technical report, Computer Graphics Laboratory, ETH Zurich, Switzerland, 2003.

[50] NVIDA. CUDA C Programming Guide.

[51] NVIDIA. Cuda home page. http://www.nvidia.com/object/cuda_home_new.html.

[52] NVIDIA. Geforce-gtx-960 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications.

[53] NVIDIA. Gpu gems 3. http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html.

[54] Oh, S., Kim, Y., and Roh, B.-S. Impulse-based rigid body interaction in sph. *Computer Animation and Virtual Worlds 20*, 2-3 (2009), 215–224.

[55] Persson, E. http://www.humus.name/index.php?page=Textures.

[56] Reid, A. Parallel fluid dynamics for the animation industry. Master's thesis, University of Cape Town, 2009.

[57] Rideout, P. Single-pass raycasting. http://prideout.net/blog/?p=64.

[58] Rost, R. J. *OpenGL(R) Shading Language (2Nd Edition).* Addison-Wesley Professional, 2005.

[59] Rusinkiewicz, S., and Levoy, M. QSplat. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (New York, New York, USA, 2000), ACM Press, pp. 343–352.

[60] SCHARSACH, H. Advanced gpu raycasting. In *In Proceedings of CESCG 2005* (2005), pp. 69–76.

[61] SCHECHTER, H., AND BRIDSON, R. Ghost sph for animating water. *ACM Trans. Graph. 31*, 4 (July 2012), 61:1–61:8.

[62] SOLENTHALER, B., AND PAJAROLA, R. Predictive-corrective incompressible sph. *ACM Trans. Graph. 28*, 3 (July 2009), 40:1–40:6.

[63] SOLENTHALER, B., AND PAJAROLA, R. Predictive-Corrective Incompressible SPH. *ACM Transactions on Graphics 28*, 212 (2009), 1–6.

[64] STAM, J. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[65] THRANE, N., AND SIMONSEN, L. *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing.* Masters thesis, Aarhus University, 2005.

[66] THUEREY, N. *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method.* PhD thesis, Universität Erlangen-Nürnberg, 2007.

[67] VAN DER LAAN, W. J., GREEN, S., AND SAINZ, M. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 91–98.

[68] VINCE, J. *Essential Computer Animation Fast: How to Understand the Techniques and Potential of Computer Animation*, 1st ed. Springer-Verlag, London, UK, UK, 1999.

[69] VINES, M., LEE, W.-S., AND MAVRIPLIS, C. Computer animation challenges for computational fluid dynamics. *International Journal of Computational Fluid Dynamics 26*, 6-8 (July 2012), 407–434.

[70] VOLKOV, V. Better performance at lower occupancy. http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf, 2010.

[71] WESTOVER, L. A. *Splatting: A Parallel, Feed-forward Volume Rendering Algorithm.* PhD thesis, Chapel Hill, NC, USA, 1991. UMI Order No. GAX92-08005.

[72] WILT, N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.*

[73] WITKIN, A., AND BARAFF, D. Physically based modeling: Principles and practice. *SIGGRAPH 2001 Course Notes* (2001), 28.

[74] WYMAN, C., AND DAVIS, S. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 153–160.

[75] XUE, D., AND CRAWFIS, R. Efficient Splatting Using Modern Graphics Hardware. *Journal of Graphics Tools 8*, 3 (Jan. 2003), 1–21.

[76] ZHANG, F., HU, L., WU, J., AND SHEN, X. A sph-based method for interactive fluids simulation on the multi-gpu. In *Proceedings of the 10th International Con-*

ference on Virtual Reality Continuum and Its Applications in Industry (New York, NY, USA, 2011), VRCAI '11, ACM, pp. 423–426.

[77] Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers on - SIGGRAPH Asia '08* (New York, New York, USA, 2008), ACM Press, p. 1.

[78] Zhu, Y., and Bridson, R. Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July 2005), 965–972.

[79] Zwicker, M., Pfister, H., van Baar, J., and Gross, M. Surface splatting. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01* (2001), 371–378.