# GPU Accelerated Particle Methods for Simulating and Rendering Fire and Water Effects

A thesis presented in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Albany
New Zealand.

Timothy Lyes

2015

**Abstract**

The simulation of complex natural phenomena such as fire and water is a complicated problem and with the surge in popularity of video games and other interactive media, it has become an area of interest in computer graphics to be able to simulate these phenomena in real-time. The challenge exists not only to simulate as accurately as possible for the best degree of visual realism, but also to use a method which allows for this real-time interaction.

In this thesis, the use of particle systems as a method for simulating fire and water effects is explored, as well as the rendering methods used to visualize them. Particle systems are well suited to this type of problem as they can be parallelized and provide many methods of behavioural customization in order to produce a wide range of different effects. Realistic looking results can be achieved when a sufficient number of particles are able to be simulated within an adequate time frame.

It can be shown that particle system methods such as Smoothed Particle Hydrodynamics and Velocity-Vortex methods are able to simulate these phenomena well. These methods are implemented using NVIDIA CUDA to parallelize the governing algorithms on the graphics processor, and with the use of spatial grid division techniques to reduce the computational complexity, they are able to run at real-time interactive rates.

Additionally, when utilizing point-based approaches for rendering fire, and a surface generation approach using the Marching Cubes algorithm for rendering water, it can be shown that these particle systems are able to be rendered with realistic-looking visualizations while maintaining interactivity. Combining both the computational aspects of the particle system and the rendering aspects directly on the graphics device produces good quality rendered fire and water effects at speeds fast enough to be used with interactive media applications.

**Acknowledgements**

I would like to thank everyone who has supported me. In particular I would like to thank my parents, Joanna and Rex, and my two sisters, Amy and Susannah, who have continued to love and support me during this endeavour. I would also like to thank my supervisors, Dr. Arno Leist, Dr. Daniel Playne and Prof. Ken Hawick for their guidance and expertise, as well as all my colleagues at Massey University for their camaraderie and encouragement.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

This thesis explores the simulation of complex natural phenomena with real-time interactivity. Complex natural phenomena can be thought of as any sort of effect or behaviour which might occur often thoughout everyday life, but due to an inherently complicated cause. Typically this comes in the form of aspects of nature - fire, water, snow, rain are many different examples though it does not necessarily need to be restricted to "natural" effects. Simulating an explosion, for example, is not a natural occurrence, but the behaviour of an explosion can only be predicted to a certain degree, and many other natural factors (air, wind, etc...) may also need to be taken into account when performing this prediction. The fact that such a large number of aspects can influence the behaviour of these natural phenomena is what makes these effects so interesting to try and predict. Highly complex behaviours (sometimes even fractal) are frequently observed in nature, but recreating this behaviour for ourselves can be troublesome.

Physical science and chemistry has provided an understanding of chemical composition of substances down to the atom, as well as intermolecular interactions within substances and chemical reactions between different substances. It should be possible to easily predict these effects since we know exactly what they are made of and how they should behave. The problem is that the amount of information needed to accurately perform these predictions is absurd; Simulating a single matchstick flame would require countless numbers of atoms. To simulate the burning of some fuel source, the complete chemical make-up and structure of the fuel would need to be provided. In most cases, performing such a simulation at such a small scale is just not practical.

This is often what is known as simulating on a *micro* scale. Perhaps in the future with significantly better computational resources than we have now, it may be possible to accurately predict these things at such a scale. Until then, it must suffice to simulate these phenomena on a *macro* scale. A macro scale does not necessarily specify an exact scale, but simply some arbitrary scale which is larger than a micro one. Simulating behaviour on a macro scale basically means that the simulation should be accurate at least from the perspective of the observer - if one were to look more closely however, the simulation would not necessarily model the

intricacies of the phenomena on a smaller scale. This typically involves making assumptions about certain aspects of the phenomena being simulated in order to simplify the sets of rules one would need to perform an accurate prediction. Thus it becomes possible to accurately predict the *macro* behaviour of a phenomena using these rules. This is done in everyday life more often than one would expect; The average driver does not need to understand the complete inner workings of every component in a car to be able to predict how the car will move if the accelerator pedal is pressed. The same is true for natural phenomena simulations - it doesn't necessarily have to perfectly explain a phenomena down to the atomic level for the simulation to be useful or provide understanding or prediction of the effects.

An important factor in choosing the levels of complexity for simulation models is time. The more complex or lower scale the simulation is, the more time it will take to complete. For highly accurate complex simulations, rendering even a single frame can sometimes take hours or even days - in the movie industry this rendering time is reasonably commonplace. As a result there has been much research into methods which help speed up this rendering time, while maintaining movie-quality representations of the effects themselves. Through the ever-growing power of technological devices, these methods of rendering have become faster and more complex, allowing the simulation and rendering of effects to become more and more realistic.

*Real-time* rendering is a different matter. Post-processing techniques seek to optimize the rendering time of a predefined simulation or effect. For real-time rendering on the other hand, it is the rendering time which is predefined. Real-time means that frames are generated and rendered at the same time as the previous frame is being displayed, and these frames are processed by the viewer at a rate as though they were seeing the effects in real life. Simulations which are implemented with real-time rendering must often compromise on the quality of the simulation itself, as more complex and physically accurate methods would simply take too long to be able to be computed and rendered at an interactive rate.

Real-time rendering is particularly important in video gaming and other interactive media. Interactivity is the driving force behind much of the research in the video games industry relating to graphics, and it is a huge component in the realism and immersion aspects in these games. Visual representations of fire in video games are a common but excellent example of this. In older games, fire effects could be created by simply repeating a small movie-sprite effect on screen with the shape and behaviour of a flame. This is sufficient, for example, if the player is far away from the effect. However, if the player moves closely past the flame effect, they would expect the flame to behave accordingly, but a predefined movie-sprite would not do this. Alternatively, the game could implement an additional animation to be played in the event of a player interaction. While this would be a more interactive case, the animation being played might not be realistic in response to the player's movements. In the case of a puddle of water, for example, a player might step in the puddle and a ripple animation may be played, but little or no water would be displaced. All these little details add to the player's immersion in the game and while not necessarily *essential* for a fulfilling gameplay experience, they are something to strive for especially as the possibilities for the incorporation of these effects become more

and more available with the steadily increasing computing power of consumer-grade graphics cards.

This thesis delves into the topic of real-time rendering in the context of the two different types of natural phenomena - fire and water. These two effects in particular were chosen for the most part simply because these are some of the most common effects one would see used in the video and movie industries. However, these effects are also interesting because of how different these effects are from each other, both in how they behave physically and appear visually.

## 1.1 Aim and Research Methods of the Thesis

Particle system methods are a subset of methods that can be used for the simulation of fire and water phenomena, and will be the methods which this thesis will focus on. A thorough introduction to particle systems will be given in Chapter 2. For now, particle systems can be considered simply a collection of small elements following some set of rules. They are particularly intriguing especially when relating back to the idea of macro and micro level simulations - whether it is practical for particles to be made small enough to be physically representative of an atom. This is not yet a practical scenario, but it makes it quite easy to make that connection and envision how groups of particles could then make up a larger physical phenomena or effect. Functionally, particle systems are highly interactive which is perfect in a real-time rendering context. However, they take a lot of computation to be able to model these effects realistically. This leads into the secondary focus of this thesis, the parallelization of these methods. Particle methods lend themselves well to parallelization so a great deal of computational speed-up should be able to be obtained through parallelization, which will help with the goal of real-time interactivity.

The work in this thesis considers the following questions:

1. Are particle systems able to be used to effectively simulate the behaviour of fire and water effects in real-time?

2. How do different particle models compare when simulating these effects, and what are the strengths and weaknesses of each?

3. How can the simulations produce an accurate visual representation of these phenomena within the confines of real-time execution speeds?

4. How can parallelization of these approaches be implemented to provide effective computational speed-ups?

5. What are some trade-offs between computational and rendering aspects of the simulation which will aid in achieving real-time interactivity?

Parallelization of the particle and rendering methods is done using NVIDIA's CUDA parallel computing platform on a GeForce GPU. This allows for high performance speed-ups for

code which is easily parallelizable, which the particle methods and rendering methods are expected to be. Rendering is done using the OpenGL graphics API, which shares interoperability with CUDA for further performance benefits.

In comparing and testing the results, performance results are graphed for execution times for relevant sets of CUDA kernels, as well as energy conservation graphs for comparing integration methods. When comparing visual results it is difficult to quantitatively measure realism - therefore, a visual comparison will be performed instead, looking at the visual results given by different rendering approaches and particle models while identifying key artifacts or behaviours which are generally regarded as more representative of a realistic result. These renderings will also be compared against real-life photos of these phenomena, again analyzing the differences and similarities between them.

## 1.2 Key Contributions of the Thesis

The key contributions of the thesis are as follows:

- A demonstration that particle systems can be used to effectively simulate fire and water effects in real-time, using OpenGL for graphics rendering and NVIDIA CUDA for parallelization of the algorithms.
- An implementation of parallel approaches for computational speed-up and spatial partitioning algorithms which allow these particle systems to run in real-time at interactive speeds.
- An implementation of a surface generating algorithm used in combination with an underlying particle system to simulate a free-surface representation of water at real-time speeds.
- An analysis of the strengths and weaknesses of various particle models used when simulating fire and water.
- A discussion on the optimal setups for these systems, when balancing computational complexity and realistic rendering aspects of the simulation.

## 1.3 Structure of the Thesis

The thesis has been structured into 7 different chapters including a concluding chapter.

There are three chapters introducing the relevant literature, each one focusing on a core aspect of the thesis. Chapter 2 provides an introduction into the subject of particle systems. It gives a short history of the usage of particles in video games and movies and provides different examples of current areas of research. This leads into Chapter 3, which is an introduction to graphics processing units (GPUs) and parallel programming, which is essential for programming real-time particle system applications. It provides an overview of the Compute Unified

Device Architecture (CUDA), a parallel computing platform developed by NVIDIA for general purpose programming on GPUs. The final theory-focused chapter, Chapter 4, goes into detail describing four different particles models investigated in this thesis. Two models, the non-interacting and the velocity-vortex models, are used to simulate fire. The other two - the spring-mass and smoothed particle hydrodynamics models - are used to simulate water. The strengths and weaknesses of these methods are also discussed.

The next three chapters focus on the implementation of the various methods and tools described in Chapters 2, 3 and 4. Chapter 5 covers the implementation of both the integration methods and the spatial grid. These parts provide the building blocks upon which the specific particle models are built. Choices in integration methods affect the viability of simulating in real-time, while also playing a huge role in whether or not the particle system remains stable throughout the simulation. The spatial grid is a parallelization tool used to reduce the number of particle-to-particle interactions so that the particle methods are not so computationally expensive. Chapter 6 goes into the core implementation of the four particle models using CUDA. Each model implementation is described through algorithms and code fragments, and each of the two pairs of models representing fire and water are compared, discussing the challenges and lessons learned in implementing them. Finally, Chapter 7 explores the different ways fire and water can be rendered. This includes point-based methods for fire and surface generation methods for water. Surface generation is particularly interesting because it can produce some very pleasing effects when combined with Fresnel reflection and refraction mapping.

Concluding the thesis, Chapter 8 will discuss the results found in Chapters 5, 6, and 7, covering important factors in the implementation of the particle models in parallel, the implementations of spatial partitioning tools and integration methods, and also the optimization of surface generation and the trade-offs required for real-time rendering. This chapter will also provide a summary of the results and discuss some possible future work in this area.

# CHAPTER 2

# INTRODUCTION TO PARTICLE SYSTEMS

Particle systems can be used to model very complex systems fairly accurately, being interactive with the user and utilizing a wide array of rendering methods which can introduce a great deal of realism to the simulation. They are an important part of computer graphics due to their ideal representations of many different visual effects and behaviours. This chapter serves as an introductory chapter to particle system simulations and their applications.

Section 2.1 will provide a general introduction of particle systems and the way they work. Section 2.2 gives a brief overview of fluid flow and hydrodynamics, central to the work in this thesis, while Section 2.3 describes some examples of other commonly used particle system applications. In the same way, Section 2.4 briefly describes the types of rendering effects that will be featured in this thesis as well as the preferred particle systems that would use these effects. Section 2.5 describes the process of numerical integration when dealing with particle systems, while Section 2.6 talks about the time complexity of the general particle system as a whole. This leads into the parallelization of particle systems in Section 2.7 and an introduction to the next chapter, dealing with parallelization and CUDA.

## 2.1 Particle Systems Overview

Particle systems were first introduced in computer graphics research by Reeves [110] in 1983, as a method to render "fuzzy" objects. At the time the standard computer graphics techniques involved the rendering of rigid bodies and smooth surfaces, and effects such as fire, water and smoke could not be modeled in such a way. Instead, a *cloud of particles* was used to represent the volume of these objects. This allowed the objects to change and move over time without completely specifying the object's shape or form. In addition, because a particle is simpler than a standard polygon, more complex systems could be processed in the same amount of time. This method was used to generate the fire effects in the famous Genesis Demo sequence from the movie *Star Trek II : The Wrath of Khan* [110].

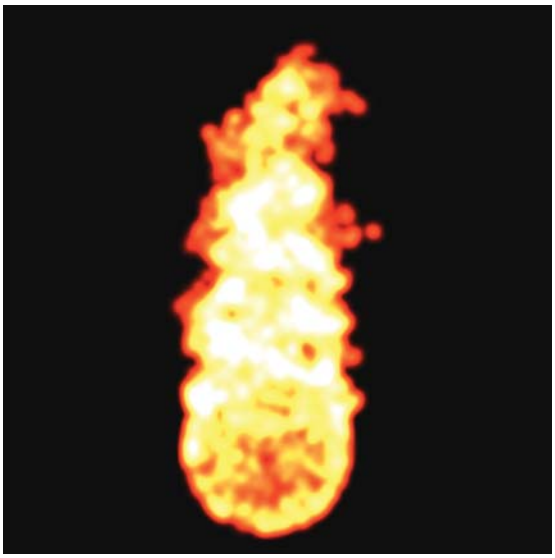

*Figure 2.1: Example screenshot of a particle system. This particle system is simulating a flame or fire and is rendered using point sprites.*

The most prominent usage of particle systems is indeed in the areas of computer simulation for movies and video games. They can be used for creating animated effects such as fire, smoke, gunfire and explosions [49] as well as fluids such as water, slime or lava [83]. More complex particle systems can be used to simulate waterfalls [118] or flocks of birds [111]. Following in the footsteps of *Star Trek II : The Wrath of Khan*, more recent motion picture productions employ particle systems for other computer generated special effects. In the movies *Lord of the Rings* and *Star Wars: Episode 1*, particle systems were used to model large crowds, while in *The Matrix Reloaded* a particle system approach to cloth simulation was used to model the protagonist's cloak in a computer generated fight scene [142]. However, particle systems can also be applied in other diverse areas of research. Goss uses a particle system approach to model ship wakes in a real-time air and sea simulation [39], while Helbing uses particle systems to model traffic dynamics [54].

Particle systems can be loosely defined as a finite set of points in space with a set of algorithmic rules determining their behaviour and appearance [49] [28]. While the system can technically exist in any dimensional space, in practice they are most often used to simulate two- or three-dimensional objects or effects. The behaviour of the system typically only refers to how a particle *moves*, which is dependent on both a direction and speed, more commonly represented as a *velocity vector*. Its appearance can refer to one or many attributes, such as a particle's *colour*, *size* or *shape*, however particular simulations may render particles as simple one-pixel grayscale point. The level of complexity of the particle system is determined by the set of algorithmic rules used to update particle properties. Many particle effects come pre-packaged in game development kits so that programmers do not have to deal with the complexity of high-quality particle system rules [49].

Ebert describes particle systems as one of five advanced modelling techniques which can

be used to simulate complex natural phenomena both controllably and efficiently [33]. Particle systems, alongside fractals, implicit surfaces, grammar-based models and volumetric procedural models, provide an abstraction to the otherwise highly complex natural model. Particle systems differ from the other four modelling techniques in that the abstraction is in the control of the simulation, where the behaviour of the particles themselves is controlled algorithmically.

When running a simulation using a particle system, initial creation of particles in a system is required. However, particles need not be destroyed if such a behaviour is desired by the programmer. Similarly, if a particle has been destroyed, the system is not obligated to *re-create* it. For example, if a particle system were used to model a single explosion - for instance a firework - the explosion event should only happen once. Therefore the particle system would require particles to be created only once at an initial position, and destroyed once they have reached their final position. In contrast, a particle system simulating rain droplets might run for an indefinite amount of time; raindrop particles would be initialized at the top of the simulation environment, fall and be destroyed as they hit the ground, then *re-created* at the top again to continue the simulation until the desired rain effect has ended.

While every particle system enables the creation and destruction of particles (if necessary), the *interaction* of particles with the environment and with each other can vary dramatically from system to system depending on the type of particle system used.

Different particle systems may fall under two main categories - a *stateless* system or a *state-preserving* system [66]. *Stateless* particle systems are the simpler of the two as they do not require particle data to be stored - rather, the particle motion is calculated based on some set of global values, such as the starting position (commonly called the *emitter*) and the current timestep. These systems can be used to model basic visual effects, but cannot be used in systems where interactions must occur; because there is no current particle data stored, the system can never know if two particles are touching each other or whether or not they are within a certain boundary.

In contrast, *state-preserving* particle systems store particle data such as *position* and *velocity*, and update this data every timestep of the simulation. This allows particle-to-particle interaction to occur, as well as imposing boundary conditions restricting particle movement (for example, particles used to simulate a fluid can be contained in a glass of water). State-preserving systems can be used in a much wider array of applications, as the vast majority of phenomena to be simulated use rules for dynamic change in the system. The focus of this thesis will be on state-preserving particle systems.

State-preserving particle systems are often used in simulations where simple laws of physics are used to govern the behaviour of the particle system *at least* at a macro level - that is, although the system might utilize physics formulas for the calculation of various particle attributes, the true chaotic nature of the system would be too complex to simulate completely and thus macro-level estimations must be made. These systems use Newtonian mechanics to guide their behaviour in the system, relying on the *mass*, *position*, *velocity* and *acting forces* on the particles to determine their motion. Acting forces can come in two different forms. *Internal forces* act on the particles from inside the system, such as electrostatic charge, or in the case

of a cloth simulation, tension or shearing forces between particles [32]. *External forces* are applied by the simulation environment and might include wind forces, whereas a gravitational force may fall into either category depending on the type of system simulated. The number of forces applied on the system will depend on how complex the programmer wants the particle system to be. The more forces applied to the system to simulate real-world effects, the more computationally expensive the simulation becomes.

It is difficult to cover all known uses of particle systems in research as there are simply too many. However, there are a few notable areas where particle systems have seen prominent use. The following sections will describe some examples of research into particle systems. Please note that each of these examples are themselves large, complex areas of research which whole theses could be written about. As such, the following sub-sections will not go into any significant detail, but merely attempt to provide an overview of the subject at a general level.

## 2.2 Fluid Flow and Hydrodynamics

Fluid dynamics is a large area of research, where different systems attempt to model the motion of fluids - including both liquid and gaseous substances - as accurately as possible. Naturally this is difficult to achieve given the inherent chaotic nature of real-world substances versus the numerical limitations of a computer simulation. *Computational Fluid Dynamics* (CFD) is a specialization of fluid dynamics which deals with the use of numerical methods to solve these problems, approximating fluid-like behaviour up to a reasonable degree of accuracy. The most famous set of equations related to CFD is the *Navier-Stokes* equations [48] [129] [19], describing the motion of viscous fluids. These equations remain the core of CFD even today.

The most common approaches to CFD are grid-based approaches, also known as *Eulerian* methods [48] or finite differencing methods. Eulerian methods describe fluid flow in terms of velocity within a grid. The Navier-Stokes equations often refer to Newton's Second Law being applied to a portion of the fluid for conservation of momentum. In the Eulerian form, the equation for the conservation of mass is given by the equation shown in Equation 2.1, while the equation for conservation of momentum in a fluid is given by Equation 2.2.

$$\frac{\delta \rho}{\delta t} + \nabla \cdot \rho \vec{v} = 0 \tag{2.1}$$

$$\rho(\frac{\delta \vec{v}}{\delta t} + \vec{v} \cdot \nabla \vec{v}) = -\nabla p + \nabla \cdot T + \rho g \tag{2.2}$$

where $v$ is the field velocity, $p$ is the pressure, $\rho$ is the fluid density, $T$ the total viscous stress tensor and $g$ the acceleration due to gravity. Many different forms of the equation can be used in various fluid dynamics simulations depending on the properties of the fluid. For instance, a popular simplification of the equation assumes the fluid to be *incompressible* and maintains

constant viscosity. Such a simulation would use the Navier-Stokes equation given in Equation 2.3.

$$\rho(\frac{\delta\vec{v}}{\delta t} + \vec{v} \cdot \nabla\vec{v}) = -\nabla p + \mu\nabla^2\vec{v} + \rho g \tag{2.3}$$

where $\mu$ refers to the constant viscosity of the fluid.

Examples of Eulerian approaches to CFD include the Lattice gas [59] [50] [51] [77] or Lattice Boltzmann approaches. These are highly-parallelizable cellular automata models used to solve the Navier-Stokes equations relatively quickly and to a good level of accuracy. This is done by using a two-dimensional triangular lattice structure, or on a three-dimensional projection of a four dimensional face-centered hyper-cubic lattice structure. Another approach to CFD is the shallow-water model [106], which uses a number of *layers* to approximate the behaviour of the fluid in *shallow* systems. Shallow-water models work well when modelling systems such as planetary atmosphere and weather models or for shallow oceanic problems, but perform poorly when trying to model any kind of turbulent behaviour. The shallow-water model is also highly parallelizable.

An alternative method to the grid-based Eulerian approach is the *particle-based* Lagrangian approach [84] [86]. As the description suggests, *Lagrangian* approaches divide the fluid up into particles and apply the fluid flow equations to the movements of each particle in the system. The Navier-Stokes equations take a different form over the Eulerian form when using Lagrangian approaches, as shown in Equation 2.4.

$$\frac{\delta\vec{v}}{\delta t} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot T + g \tag{2.4}$$

Lagrangian methods simplify the Navier-Stokes equations in two ways [87]. Firstly, particle systems are guaranteed to maintain conservation of mass, because each particle has a constant mass. Therefore, the conservation of mass equation shown in Equation 2.1 can be ignored. Secondly, because particles move with the fluid, the derivative of the velocity field is the equivalent of the derivative of the particles with respect to time. This means that the convective term $(\vec{v} \cdot \nabla\vec{v})$ is also not needed [87].

Eulerian methods are typically more accurate than Lagrangian, but with the disadvantage that the fluid must stay confined within the grid. Additionally, it is very difficult for grid-based approaches to permit interactivity with the fluid. In contrast, Lagrangian methods provide approximations of field functions from particles in any arbitrary location, while the approximations become more accurate as the density of the particles increases. Stam proposed a semi-lagrangian approach to solving the Navier-Stokes equations for a more stable fluid simulation [119].

In general, examples of fluid phenomena are commonly water or air related - waves and

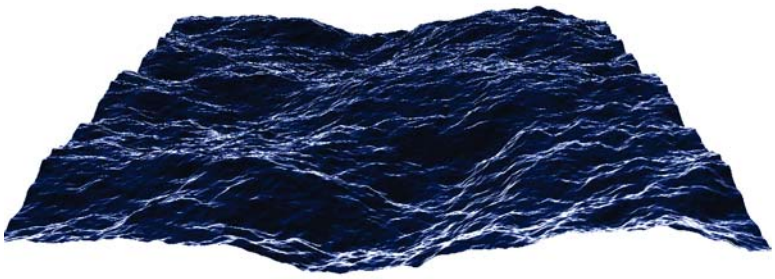

*Figure 2.2: Example screenshot of using a fast fourier transform (FFT) approach for wave simulation using CUDA, featured in the CUDA Toolkit and based on the work of Tessendorf [128].*

weather patterns are often simulated, but simple water in a pool or glass is also important in the area of video games and movies and can add a great deal of realism if the simulation or animation is of a high quality. Bridson [17] provides several methods of animating fluids including fire, water and smoke. While most particle systems share a high degree of complexity, fluid particle systems stand out in terms of complexity as attributes such as *convection*, *diffusion*, *turbulence* and *surface tension* [87] need to be taken into account if the simulation is to be believable. Because of this extreme level of complexity, realistic fluid particle systems are more often than not rendered in non-real-time. However, recent advances in parallel computing technologies have enabled some very advanced and highly realistic real-time fluid simulations to be produced [144].

Some fluid flow simulations often involve introducing additional materials into the flow - for instance, the introduction of dyes or oil into a glass of water will produce some sort of reaction, which is interesting to visualize. Other variations of fluid flow involve the simulation of waves. A wave simulation could be considered a type of surface modelling problem as we are only interested in visualizing the top surface of the system, not the underneath the surface. One of the most common methods of wave simulation is actually a fast fourier transform approach for building *height fields* (see Figure 2.2), first introduced by Tessendorf [128]. This method has been widely adopted in the video game and movie industries for wave simulation. However, particle system approaches to wave simulation also exist, such as Kass and Miller's computationally inexpensive method of rendering fluids, specifically looking at wave motion and reflection [61]. Fluid models have even been used for simulating sand motion [146].

The computational and/or memory requirements for most fluid flow particle systems is high and it demands approaches to deal with large amounts of data very quickly. Thankfully, with the introduction of parallelization methods on Graphics Processing Units (GPUs) these simulations have the ability to be run in real-time. Krüger et al [67] provide a simulation of an incompressible fluid-flow particle system which is visualized running on a three-dimensional uniform grid. The method improves the computation and memory limitations in particle tracing by using graphics card functionality to carry out particle updates - as well as rendering

the simulation - directly on the device. Additionally, Kipfer et al. developed the *UberFlow* Particle Engine [64] designed for simulation of large particle systems in real-time on GPUs. Algorithms for efficient collision detection and sorting are included in this engine. Other examples of high-quality fluid simulations include Müller et al.'s method [87] for simulating fluids based on the Smoothed Particle Hydrodynamics (SPH) [76] [85] approach. Zhang et al. have more recently developed a fluid simulation based on the SPH model using a Multi-GPU parallelization approach [144]. The multi-GPU approach, combined with fluid effect rendering techniques, produces an extremely high-quality fluid simulation. GPU methods of particle simulation as well as SPH approaches will be explored in this thesis.

Other common methods for simulating fluids include cellular automata approaches such as the Lattice gas [59] [50] [51] [77] or Lattice Boltzmann variants, used to approximate fluid flow with a good degree of accuracy using a two-dimensional triangular lattice structure. The shallow-water model [106] can be used when simulating a "shallow" fluid system, where a small number of layers are used to approximate the behaviour of the fluid as a whole.

## 2.3 Other Related Particle Models

Particles are used in a large array of different research capacities. Some of the most common usages will briefly be described in the following subsections. Any methods more relevant to this thesis will be described in greater detail in later chapters.

### 2.3.1 Agent-Based Models

Particles have seen extensive use in flocking and crowd simulation problems. In this instance, particles are referred to as *agents*, hence the name *Agent-based modelling*. Pair-wise interactions between agents are the core of this approach - particles behave based on the behaviour of the other particles in the system. The classic example of agent-based modelling for use in flocking models and group behaviours in general is Reynold's *Boids* [111], designed to simulate the behaviour of flocks of birds. The Boids model could be thought of as a more complex subset of the general particle system model. Reynolds explains one key difference which sets Boids apart from the generic particle system - Boids have an *orientation* and a full local coordinate system. Agents in the flock behave according to a few select forces.

1. *Collision avoidance*, where agents avoid collisions by exerting a repulsive force on one another.

2. *Velocity matching*, where agents try to move at a similar speed to that of their neighbouring agents.

3. *Flock centering*, where agents are drawn to the center of the flock.

These three forces or rules are in order of precedence - for example, an agent will attempt to match velocity only if in doing so it also avoids collision with another agent. Additionally,

a random *wander force* element can be introduced to give the appearance the agents have a "mind of their own" [142].

Crowd simulations in particular have seen extensive use in the movie industry. A crowd simulation program was used to produce large numbers of autonomous agents making up many of the battle scenes in the movie *Lord of the Rings: The Two Towers* [116]. Each agent may have different actions depending on the level of aggressiveness of the particular agent and the surrounding agents. These actions include deciding which part of the body to attack or defend, as well as retreating.

While particle systems usually have applications in computer graphics and visualization, flocking and crowd simulations are generally applied to more predictive problems. Agent-based models are also used when representing individual cars as agents when simulating traffic [18], while other approaches involve models designed for use with the equations of motion (Newton's Law) when predicting airbag deployment [14].

### 2.3.2 Spring-Mass Models

Spring-mass models are physics-based models that involve *pair-wise interaction* between particles by connecting them with *springs*. The springs exert forces on each particle depending on the distance between these particles and this is what determines how the system behaves. The size of the different forces that the springs exert can be changed according to the type of material that is being simulated - highly stretchable fabric, for example, would have a very low spring tensile force, while a fabric like denim would have a very high tensile force. Spring-mass models exibit soft-body dynamic properties - this is due to the springs being able to be extended or compressed by varying lengths. This is in contrast to rigid-body dynamics, where objects are unable to be extended or compressed past the initial rest distance. An analogy would be the difference between the collision of a tennis ball against a wall (it bounces) vs. the collision of a marble against a wall (there is very little-to-no bounce).

Examples of spring-mass models include the approaches to *cloth simulation*, utilizing particle methods to represent points on a cloth with tensile and bending forces between neighbouring particles to simulate the deformation (such as stretching or scrunching) of the cloth material. Cloth simulations share many similarities with work in *surface modelling* where particles are used to simulate irregular or rapidly changing surfaces instead of volumetric effects. Both simulation methods build on similar models from the agent-based flocking simulations [111] discussed earlier, in that pair-wise interactions between particles in the cloth or surface are usually implemented. Specifically in surface modelling, the idea of an *oriented particle* can be further developed. Szeliski and Tonnesen [124] make use of oriented particles, representing small surface elements with their own local coordinate frame.

As well as the internal forces exerted on pairs of particles, external forces such as wind, gravity and bounding forces also need to be considered. With numerous forces acting on particles both internally and externally, the final acceleration of each particle is given by the superposition of these forces. Eberhardt et al. [32] also use a particle system approach to implement

*Figure 2.3: Example screenshot of a cloth simulation for a flag blowing in the wind, using a spring-mass particle model.*

a fast cloth simulation specifically to improve the cloth's behaviour with environmental forces. It is based similarly on Reynolds' boids work [111], building on a model described by Breen et al. for simulation of woven cloth using a particle-based system [15] [16].

Many cloth systems are prone to instability when applying numerical integration methods to the individual particles. This is because of the nature of the cloth materials - stretching of the material is relatively difficult while bending or scrunching the material is very easy. Implicit numerical integration methods are therefore better to use than the traditional explicit methods such as Euler or higher-order Runge-Kutta methods. Baraff and Witkin [7] came up with an implicit method for cloth simulation which allowed relatively large timesteps to be used, resulting in a much faster simulation. Cloth simulation theory can also be extended to other applications such as simulating hair [117].

### 2.3.3 Other Notable Work

There have been several other notable publications related to the use of particle systems with very interesting applications.

Hastings et al. provide insight into the use of Interactive Evolutionary Computation (IEC) to help deal with the intricacies of programming a complex particle system [49]. It focuses on the use of a method called *NeuroEvolution of Augmenting Topologies* (NEAT) as well as artificial neural networks to control particles, resulting in easier development of special particle effects as well as a wider range of effects available.

Miller and Pearce introduced a method for rendering particles as "*globules*" [83], modelling *soft* collisions between particles by decreasing the particle speed gradually as one particle collides with another. The method is described as utilizing "globular dynamics", whereby particles avoid rigid collisions with each other while incorporating viscous forces and distance-

dependent soft collisions, making it useful for modelling viscous materials such as mud, lava and slime. The method also takes a particle-particle interaction approach similar to the flock simulation in Reynold's boids [111], where particles interact with other particles only within their local environment.

Sims uses a physics-based particle system to simulate a waterfall [118] as well as using a data-parallel approach for computational speed-up. Data parallel approaches will be covered in Chapter 3.

Kajiya and Kay use a *texel structure* (texture pixels) as an alternative to particles in rendering fuzzy objects, for example fur [60]. However, they suggest that particle systems may be used in tandem with texels, generating the texel models based on the particle system. A texel (not to be confused with the cg texture element) is a three dimensional array of parameters approximating visual properties of a collection of microsurfaces.

## 2.4 Rendering Effects

The rendering of particle systems is also a well-studied topic, as the behaviour of the particles drawn as simple point pixels is typically not sufficient alone to produce a realistic looking simulation - the rendering method is what ultimately matters. Individual particle primitives must be replaced using a rendering method that produces the desired effect. For example, natural phenomena such as fire, clouds, snow and rain [33] as well as smoke, gunfire and explosions [49] will all need to be rendered differently. Different approaches to rendering particles may fall under several categories;

- *Point-based* approach, where each particle is represented by a rendering primitive such as a texture quad or point sprite. These methods usually involve a *billboarding* approach where the primitive is changed to always face the camera. They work well for phenomena where detail is not important, such as rain or snow fall.

- *Volumetric rendering* approaches, where each particle is intended to be the smallest portion of a three-dimensional volume and this entire volume must be sampled. Ray-tracing methods are common volumetric rendering approaches as well as point-based rendering with *slices*. Clouds and smoke are good usages of this rendering method.

- *Surface generation* involves the conceptual "draping" of surface over the top of the particle system. It is similar to volumetric rendering except that the entire volume does not need to be sampled, only the outer surface. This method is frequently used in rendering *fluids* such as water simulations.

Many of these methods will be explored over the course of this thesis, with more detailed descriptions as well as the advantages and disadvantages of using each method when compared with different particle simulation methods and approaches.

## 2.5 Numerical Methods and Integration

The behaviour of particles can typically be represented by one or more *differential equations*. In order to solve these equations computationally, solutions must be approximated using *numerical integration methods*. These normally continuous differential equations can be divided up into finite *timesteps*. The smaller the size of the step, the greater the approximation to the true equation solution.

The movement of particles in a system can generally be represented by a set of differential equations as a function of time. The rates of change of the position and velocity of the system are given using the standard kinematic equations of motion as shown in Equations 2.5, where $\vec{x}$ is the particle position, $\vec{v}$ is the particle velocity, $\vec{a}$ the particle acceleration and $t$ the time.

$$\begin{aligned} \frac{\delta \vec{x}}{\delta t} &= \vec{v}(t) \\ \\ \frac{\delta \vec{v}}{\delta t} &= \vec{a}(t) \end{aligned} \tag{2.5}$$

Numerical integration methods can be classified into two main categories.

- *Explicit* methods calculate new states directly from the values given by the older state or set of states - for example, a new value $y_{n+h}$ is given *explicitly* in terms of the old value $y_n$ [107]. Some explicit methods may also use multiple *intermediate* states (such as the *Runge-Kutta* methods, see Section 2.5.3).

- *Implicit* methods are much harder to implement as they involve calculations using states at the beginning of the timestep and also at the end - which is currently unknown.

Implicit methods are favoured over explicit methods when equations become *unstable* at certain large timestep sizes. Such equations are typically referred to as *stiff* equations. In general, implicit methods are more stable than explicit methods, while explicit methods are more accurate [107].

### 2.5.1 Euler Integration Method

The Euler Method is the simplest and most common numerical integration method. It involves calculating $f$, the slope of the tangent to the curve $y$, then stepping along the tangent using a step size $h$. It assumes that the following point on the tangent $y_{n+h}$ also lies on the function curve, and repeats for future points. In the general form:

$$y_{n+h} = y_n + hf(y_n, t_n) \tag{2.6}$$

where $t_n = t_0 + nh$, $n$ is the current step in the method and $h$ is the step size. As the step size decreases, the approximation to the true solution becomes more and more accurate.

The Euler Method can be applied to the particle system movement functions to approximate the particle position $\vec{x}$ and velocity $\vec{v}$ using a given time interval (step size) $h$ as shown in Equation 2.7.

$$\begin{aligned} \vec{x}_{n+h} &= \vec{x}_n + h\vec{v}_n \\ \vec{v}_{n+h} &= \vec{v}_n + h\vec{a}(\vec{x}_n, t_n) \end{aligned} \tag{2.7}$$

where $\vec{a}$ is the particle acceleration and $n$ is the current timestep in the simulation.

While the Euler method is sufficient to simulate a complex system with good visual quality, there is still a large amount of error between the original and estimated curves, and this can lead to instability in the system. Error can be reduced using higher-order integration methods. The most common of these are the *Runge-Kutta* (RK) methods. RK methods refer to a set of explicit methods based on the original Runge-Kutta method most commonly referred to as the *4th-order Runge-Kutta* or RK4. The Euler method can be seen as the first-order RK method, and the simplest RK method in the set. Higher orders of RK are more complicated and require additional resources to program, but are more accurate.

### 2.5.2 Euler-Cromer Method

The Euler-Cromer method [26] is a semi-implicit variation of the Euler method where the position is calculated by the velocity at the end of the timestep, while the velocity is calculated using the standard Euler method. This is shown in Equation 2.8.

$$\begin{aligned} \vec{v}_{n+h} &= \vec{v}_n + h\vec{a}(\vec{x}_n, t_n) \\ \vec{x}_{n+h} &= \vec{x}_n + h\vec{v}_{n+h} \end{aligned} \tag{2.8}$$

The Euler-Cromer is a *symplectic* integrator [81], where the integrator works better than standard integrators when solving the differential equations of Hamiltonian systems - that is, a system where the state is described completely in terms of the *Hamiltonian*, a scalar function of the position and momentum at some timestep $h$. Symplectic integrators preserve integral invariants of these systems and thus provide a better representation of the system dynamics over long periods of time [81].

There also exists an improved Euler-Cromer method similar to the Leapfrog method discussed later. While the standard Euler-Cromer uses the velocity $\vec{v}_{n+h}$ to calculate the subsequent position $\vec{x}_{n+h}$, the improved Euler-Cromer uses an average of the two velocities $\vec{v}_n$ and $\vec{v}_{n+h}$. This results in a more accurate estimation of $\vec{x}_{n+h}$, at the cost of slightly more storage for the extra $\vec{v}$ value. Essentially this is almost the same method as the Leapfrog method dicussed

later, except the velocity timestep is simply shifted half a timestep to the right. The standard Euler-Cromer method was chosen over this improved method as a means of demonstrating the differences between a first-order non-symplectic method (Euler), first-order symplectic method (Euler-Cromer) and 2nd-order symplectic (Leapfrog).

### 2.5.3 Runge-Kutta (2nd Order)

The *second-order Runge-Kutta* method (RK2) (or simply the *midpoint* method) involves evaluating the function $f$ at the midpoint of the time interval rather than only at the beginning, and then using the midpoint slope across the entire interval width. Equation 2.9 shows the general form.

$$\begin{aligned} k &= f(y_n + \frac{h}{2} f(y_n, t_n), t_n + \frac{h}{2}) \\ y_{n+h} &= y_n + hk \end{aligned} \tag{2.9}$$

An evaluation of the function is performed by calculating the rates of change (*k-value*) of the position and the velocity at the midpoint, using these as the final rates of change across the entire timestep. Similar to the Euler method, RK2 can be applied to the particle system movement functions to approximate the particle position and velocity using a given time interval $h$ as shown in Equation 2.10. The k-value of the position is given by $kx$ while the k-value for the velocity is given by $kv$. Note that the function $\vec{a}$ determines the value for $kv$ based on the position and the velocity at the midpoint.

$$\begin{aligned} kx &= \vec{v}_n + \frac{h}{2}\vec{a}(\vec{x}_n, t_n) \\ kv &= \vec{a}(\vec{x}_n + \frac{h}{2}\vec{v}_n, t_n + \frac{h}{2}) \\ \vec{x}_{n+h} &= \vec{x}_n + hkx \\ \vec{v}_{n+h} &= \vec{v}_n + hkv \end{aligned} \tag{2.10}$$

### 2.5.4 Runge-Kutta (4th Order)

Fourth-order Runge-Kutta (RK4) is the most frequently used order of RK because it produces highly accurate results. All other orders of RK are based on a generalization of the RK4 method. It involves evaluating the function $f$ four times and producing a solution using a *weighted average* of the four results. RK4 is the highest order of numerical integration used in this thesis.

Given a step size $h$ and time $t$, the four evaluations in a general form are as shown in Equation 2.11.

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{h}{2}k_1, t_n + \frac{h}{2}) \\
k_3 &= f(y_n + \frac{h}{2}k_2, t_n + \frac{h}{2}) \\
k_4 &= f(y_n + hk_3, t_n + h)
\end{aligned}
\tag{2.11}
$$

Here $k_1$ refers to the evaluation at the beginning of the interval, $k_2$ and $k_3$ are midpoints in the interval, and $k_4$ refers to the end point. Finally, the weighted average can be found by favouring the two midpoint rates as shown in Equation 2.12.

$$
y_{n+h} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \tag{2.12}
$$

This can be applied to the particle system in a similar manner as the RK2 and Euler methods. Because there are four extra rates to be stored, extra memory must be allocated for each of the four evaluations. The rates of change for each evaluation can be shown in Equation 2.13.

$$
\begin{aligned}
kv_1 &= \vec{a}(\vec{x}_n, t_n) \\
kx_1 &= \vec{v}_n \\
kv_2 &= \vec{a}(\vec{x}_n + \frac{h}{2}kx_1, t_n + \frac{h}{2}) \\
kx_2 &= \vec{v}_n + kv_1\frac{h}{2} \\
kv_3 &= \vec{a}(\vec{x}_n + \frac{h}{2}kx_2, t_n + \frac{h}{2}) \\
kx_3 &= \vec{v}_n + kv_2\frac{h}{2} \\
kv_4 &= \vec{a}(\vec{x}_n + hkx_3, t_n + h) \\
kx_4 &= \vec{v}_n + hkv_3
\end{aligned}
\tag{2.13}
$$

Finally, the positions and velocities of the particles are given using the following Equations 2.14.

$$
\begin{aligned}
\vec{x}_{n+h} &= \vec{x}_n + \frac{1}{6}(kx_1 + 2kx_2 + 2kx_3 + kx_4)h \\
\vec{v}_{n+h} &= \vec{v}_n + \frac{1}{6}(kv_1 + 2kv_2 + 2kv_3 + kv_4)h
\end{aligned}
\tag{2.14}
$$

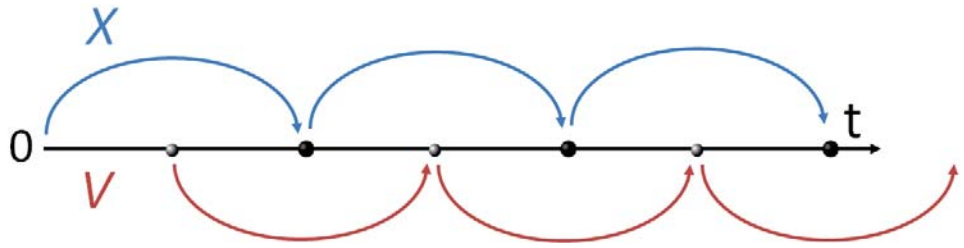


*Figure 2.4: The leapfrog method. Position increases at the same size time intervals as the velocity, however the velocity interval is staggered half a timestep to the right.*

### 2.5.5 Leapfrog Method

The *Leapfrog* integration method is similar to the *velocity verlet* method. Verlet integration methods represent particle data differently than methods such as the Euler or RK methods, where the standard Verlet integration method determines the rate of change of the position based on the *current* and *previous* positions rather than the position and the velocity. The *velocity verlet* [80] algorithm is more commonly used variant of the verlet method, where velocities are instead calculated by the current and previous accelerations. The velocity verlet algorithm is shown in Equation 2.15.

$$\begin{aligned} \vec{x}_{n+h} &= \vec{x}_n + \vec{v}_n h + \frac{1}{2}\vec{a}(\vec{x}_n, t_n)h^2 \\ \vec{v}_{n+h} &= \vec{v}_n + \frac{\vec{a}(\vec{x}_n, t_n) + \vec{a}(\vec{x}_{n+h}, t_n + h)}{2}h \end{aligned} \tag{2.15}$$

The Leapfrog algorithm is very similar, the only difference being that the positions and velocities are not calculated at the same timesteps, rather the position is calculated every timestep while the velocity is calculated a half-timestep later - i.e. they "leap" over each other. The Leapfrog method equations are show in Equation 2.16, while Figure 2.4 demonstrates the "leaping" behaviour of the method.

$$\begin{aligned} \vec{x}_{n+h} &= \vec{x}_n + h\vec{v}_{n+\frac{h}{2}} \\ \vec{v}_{n+\frac{h}{2}} &= \vec{v}_{n-\frac{h}{2}} + h\vec{a}(\vec{x}_n, t_n) \end{aligned} \tag{2.16}$$

Similar to the Euler-Cromer method, the Leapfrog (and velocity verlet, they are essentially the same algorithm) is a symplectic integrator and provides stability in particle simulations where the volume and energy should be conserved. A slight modification to the Euler-Cromer method will actually result in the Leapfrog method - evaluating the velocity at timestep $h/2$ earlier. Since the Leapfrog method is a second-order integration method, it provides better accuracy than the Euler-Cromer while still needing only a single evaluation, unlike the second-order RK2 method.

When implementing these integration methods with the particle systems, considerations

need to be made when dealing with parallelization and the complexity of the particle system. If the particle system is too complex, using a higher-order integration method may mean the system will be unable to operate in real-time using the same amount of particles. However, this means that if a lower-order integration method is used, more error will be introduced into the system. The potential advantages and disadvantages need to be weighed before making this decision. The implementation of the integration methods their advantages and disadvantages will be discussed in greater detail in Chapter 5.

## 2.6 Computational Complexity

For a system of N particles, the traditional level of computation required has been of the order $O(N^2)$. That is, as the number of particles in the system increases, the computation time required increases exponentially. There has been significant research in the areas of improving this. Most notably, *Particle-In-Cell* (PIC) methods have achieved a lot of success in decreasing the complexity. PIC approaches involve applying an abstract grid or mesh to the particle system and performing calculations on particles using the force values computed for each cell in the mesh or grid. Using this type of approach the complexity of these particle systems reduces to $O(N + M \log M)$ where $M$ refers to the dimensions of the mesh or grid. These dimensions are typically proportional to the number of particles in the system, but it is never larger than the number of particles N. In practice, the complexity of PIC methods can be reduced to $O(N)$ [43] when the number of particles in each cell in the grid or mesh is roughly the same (a uniform distribution). However, this highlights the weakness of this method, whereby systems with a highly irregular or non-uniform particle distribution will suffer greatly in performance.

*Particle-mesh* methods ($P^3M$) are a variant of Particle-in-Cell methods whereby interactions between particles in each cell are handled individually rather than using the computed force for the entire cell. This improves on the accuracy of the PIC method, though it still suffers performance degradation when systems are irregularly distributed. Additionally, greater care is needed when determining the dimensions of the grid for $P^3M$ methods, as a larger cell size will result in an exponential increase in the number of particle-to-particle interactions needing to be calculated. In contrast to a standard $P^3M$ method, Appel [4] developed a method using center-of-mass approximation to compute forces over large distances, achieving substantial speed-up where particles are spaced at *great distances* apart, becoming less efficient the closer the particle distribution gets to uniform.

## 2.7 Introduction to Parallelization

The positions, velocities and forces acting on particles are all calculated per-particle, and since the algorithms which are used to calculate these properties should be exactly the same from particle to particle, it is obvious that particle approaches follow the Single Instruction Multiple Data (SIMD) parallel architecture and are well suited for highly parallel computation [118]. Instructions or rules of behaviour are described as if addressing a single particle, but they are

applied to all particles (or a subset of them) in parallel. Parallelizing the algorithms governing the behaviour of a particle system can be done using the *Graphics Processing Unit* (GPU). In order to parallelize the particle systems used in this thesis, the Compute Unified Device Architecture (CUDA) was used. In Chapter 3 an overview of CUDA will be presented as well as an explanation of some parallelization approaches to particle systems.

# CHAPTER 3

# INTRODUCTION TO PARALLELIZATION

A *Graphics Processing Unit* (GPU) can be generally defined as a card or circuit which is developed specifically for accelerating graphics processes. Nowadays, GPUs are most often used in generating visual effects for video games in great detail. Over recent years, the demand for higher and higher rendering quality at an acceptable speed has caused a dramatic growth in GPU technology, resulting in an evolution into a highly parallel, multi-cored and multi-threaded processor. GPUs now have great computational processing power as well as very high memory bandwidth [91]. GPUs are particularly well-suited for data-parallel problems - problems wherein the same instruction is executed on multiple data elements in parallel. Due to this ability for efficient parallel execution of algorithms with high degrees of performance speed-up, as well as the fact that they have become flexible and are not restricted to graphics-only problems, GPUs have been seeing use as a *general purpose computing platform*. This is what is known as general purpose programming on GPUs or *GPGPU* [65]. Although graphics rendering was originally the sole purpose of GPUs, nowadays the design decisions for new graphics cards are influenced by the modern applications' need of a programmable pipeline which lends itself for both graphics *and* non-graphics applications.

In order to understand the evolution of GPUs and the relationship between the usage of GPUs in rendering objects or images visually and the usage of GPUs for solving non-graphical problems or controlling simulations, it is best to first understand the *graphics rendering pipeline* which all or most graphics hardware was built to employ. The rendering pipeline refers to a set of steps used to project a three-dimensional scene onto a two-dimensional screen or image using a *raster* or grid of pixels. Graphics hardware was originally designed with a *fixed-function* graphics pipeline in mind, using Graphics *Application Programming Interfaces* (APIs) to provide an interface for the programmer to use graphics libraries to send commands from the CPU to the graphics hardware for processing [65]. The two main graphics APIs used were Microsoft's *Direct3D* graphics API and the open-standard graphics API *OpenGL*. These two APIs remain some of the most well-known graphics APIs today.

Because the GPU is efficient in processing large amounts of floating point operations, the

GPU quickly surpassed the CPU in terms of raw processing power. Because of this, the use of GPUs to solve more computationally expensive problems became hugely popular in science and engineering fields. However, programming the GPUs to handle these problems proved difficult because the GPUs had been designed specifically for computations on large numbers of graphics primitives. To access the computational resources, programmers had to represent the problem as a graphics API program so that it could be launched on the GPU through API calls from Direct3D or OpenGL. Usually this was done through the pixel shader, storing data into texture images and being outputted as a set of pixels generated from the raster operations [65]. Despite the difficulties involved in using graphics APIs to solve general purpose problems, the performance speed-up achieved was substantial and as such resulted in a great demand for a programming model more suited for general purpose programming problems. This is what motivated NVIDIA to develop the Compute Unified Device Architecture, or *CUDA* [91]. CUDA provides access to the parallel programming capabilities of the GPU hardware without the limitations of using shader APIs.

It should be mentioned that the Open Computing Language API (OpenCL) can also be used as an alternative to CUDA for GPGPU computing. OpenCL differs from CUDA in that it is able to be implemented for other graphics cards (for example AMD) and multi-core CPUs, while CUDA is intended for only NVIDIA GPUs. OpenCL has OpenGL interoperability similar to CUDA, and performs equally well - at least for optimized code. However when optimizing systems for GPUs, it becomes necessary to take into account the specific architecture of the GPU which is being used. Optimized code for an AMD graphics card will not necessarily be optimized for an NVIDIA graphics card. The work in this thesis seeks to produce optimal performance results rather than good results over a range of devices, so device independence is not important in this situation. Nevertheless, this work intends to demonstrate the ideas of the algorithms in general, and while CUDA is being used to implement these algorithms, that is not to say they may not also be realised in OpenCL.

This chapter provides an overview of the CUDA programming model as well as the general parallelization approach to particle systems. Section 3.1 gives a description of the CUDA Programming Model, Section 3.2 describes the CUDA architecture and Section 3.3 explains the different memory types in CUDA and how they are used. Section 3.4 explains the general method of parallelizing particle systems for achieving good computation speed-ups. Finally, Section 3.5 gives an overview of the different CUDA libraries used for the work in this thesis and Section 3.6 briefly describes the graphic card chosen.

## 3.1 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a general-purpose parallel computing architecture, developed by NVIDIA to solve computationally expensive problems more efficiently than on the CPU. It is designed to overcome the challenge of developing application software that scales its parallelism seamlessly as the number of processing cores constantly increases, while at the same time being similar to standard programming languages such as C [91].

```
//Calculate the kinetic energy of a particle
__global__ void calc_kinetic_energy(
           float* energy,         //output - kinetic energy
           float particle_mass,   //input - mass of particle
           float4* vel,           //input - particle velocities
           uint numParticles){    //input - total no. of particles

    //Calculate thread id
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if(index >= numParticles)return;

    //Assumes all particle mass is equal and constant
    float mass = particle_mass;
    float4 velocity = vel[index];

    //Energy deals with magnitude of velocity
    float vel_mag = length(velocity);

    //E_k = 1/2 mv^2
    energy[index] = 0.5f * mass * vel_mag * vel_mag;
}
```

*Listing 3.1: An example CUDA kernel, calculating the kinetic energy of a particle.*

The NVIDIA CUDA programming model was created for developing applications for this platform. When working with CUDA, the CPU and the main memory are referred to as the *host*, which executes functions on one or more *devices* which are the GPUs. A CUDA program consists of one or more data-parallel functions called *kernels* that are executed on either the host or the device. Typically, code that is difficult or simply impossible to parallelize is executed on the host, while code which can be parallelized effectively is executed on the device. The CUDA program is compiled using the NVIDIA C Compiler (nvcc).

CUDA devices operate on a Single Instruction Multiple Data (SIMD) model, meaning that many processors perform the same program or set of instructions (the kernel) on different portions of the data. Kernels are executed across multiple *threads*, each thread being given a ID or index based on its position in the overall *grid*. This grid is a one-, two- or three-dimensional conceptual grid comprising of multiple *thread blocks*, each thread block made up of multiple threads. Thread blocks have their own block ID, which along with the grid and thread block dimensions, is used to calculate the IDs of each thread in the grid. The maximum number of threads within a thread block varies depending on the architecture used. Each thread has its own *local memory* and registers [95].

Kernels are defined using either the `__global__` or the `__device__` declaration. Global kernels may be called from the host code, while device kernels may only be called by global kernels or other device kernels - they cannot be called by the host. Kepler architectures allow the execution of up to 32 *concurrent kernels*. An example kernel is shown in Listing 3.1 relating to the calculation of the kinetic energy of a particle.

The threads inside a thread block execute concurrently and can communicate with each other via *shared memory*. Threads using shared memory need to be careful not to access shared data which is currently being used by another thread in the same block. To avoid this,

*barrier synchronization* can be specified by the programmer using the `__syncthreads()` function. A single thread block will execute on a single *streaming multiprocessor* [91] (SM, or SMX in Kepler architecture) - this is how threads are able to cooperate via shared memory and barrier synchronization. All blocks within the grid may also access *global memory*, but access to this memory is slower than shared memory as global memory is stored off-chip. Similarly to when accessing shared memory, global memory access requires threads to be synchronized if multiple threads intend on accessing the same data which may potentially be modified. For a better description of CUDA memory types, see Section 3.3.

Streaming multiprocessors execute threads in groups of 32 threads called *warps*. Thread blocks are partitioned into warps containing threads of consecutively increasing thread IDs. Each thread is mapped directly to a CUDA core and individual threads composing a warp start together at the same program address. Threads inside a warp are allowed to diverge through data-dependent conditional branches (for example if statements or loops). Branches are executed serially by every thread and are masked by CUDA depending on what conditional branch each thread follows. The execution masking means that threads which do not follow a certain branch perform the equivalent of a null operation instead. Branch divergence means that if either branch is needed to be executed by at least one thread in a warp, then both branches will be run for all threads in the warp, which is inefficient and will result in sub-optimal performance. For this reason, divergence within warps should be minimized wherever possible.

## 3.2 Kepler Architecture

GPUs using the Kepler architecture exceed the computational power of the previous generation Fermi cards, as well as doing so more efficiently with less power consumption [95]. Kepler architecture comprises of two major variants, the GK104 and GK110 [93]. The GK104 and GK110 support CUDA Compute Capability 3.0 and 3.5 respectively, and the GK110 variant is the architecture used for the work in this thesis. Table 3.1 shows the statistics for the Kepler GK110 architecture.

| | |
|---|---|
| Compute Capability | 3.5 |
| Threads / Warp | 32 |
| Cores / SM | 192 |
| Max Warps / Multiprocessor | 64 |
| Max Threads / Multiprocessor | 2048 |
| Max Thread Blocks / Multiprocessor | 16 |
| 32-bit Registers / Multiprocessor | 65536 |
| Max Registers / Thread | 255 |
| Max Threads / Thread Block | 1024 |
| Max X Grid Dimension | $2^{32}$-1 |
| Dynamic Parallelism | Yes |

*Table 3.1: Kepler GK110 architecture statistics.*

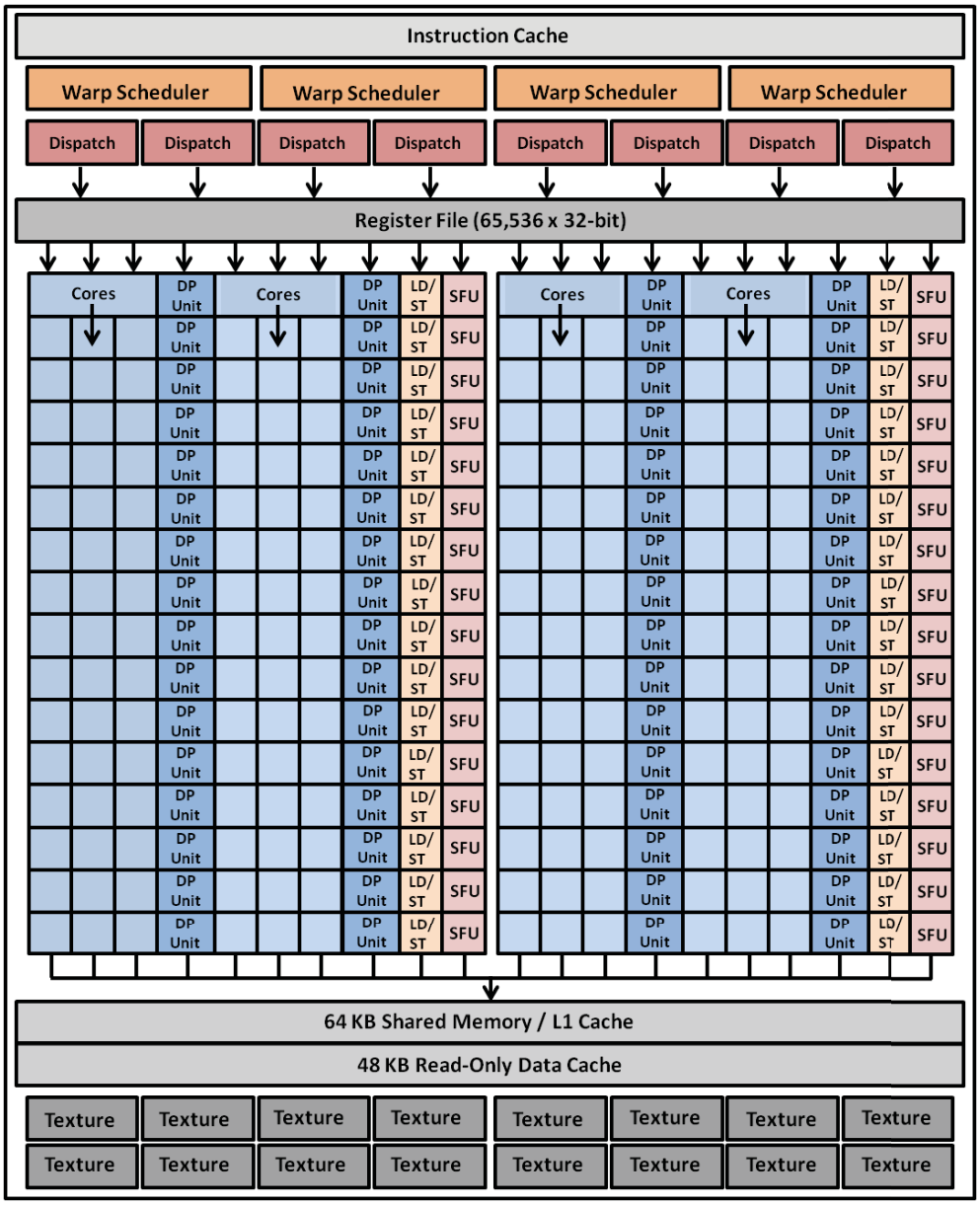


*Figure 3.1: Visual representation of the Kepler SMX layout.*

The Kepler architecture introduced several new improvements on the Fermi architecture design [95].

- *The SMX processor architecture* features 192 single-precision CUDA cores. SMX utilizes the primary GPU clock as opposed to the 2x shader clock which was used all previous generations. This was due to an overall goal of the GK110 design - better performance with respect to power consumption. Although the GPU clock runs at a lower speed, this is made up for by the increase in the total number of processing cores. The GK110 architecture can have up to 15 SMXs. A visual representation of the Kepler SMX processor is shown in Figure 3.1 [95].

- *The quad warp scheduler* schedules warps to be issued and executed in batches of four. Kepler implementations allow double-precision instructions to be paired with other instructions.
- The GK110 provides up to *255 registers per thread*, resulting in better performance for codes that would otherwise suffer spilling behaviour due to an insufficient amount of registers per thread.
- *Shuffle instructions* allow threads within a warp to share data faster than shared memory.
- *Atomic operations* have been improved with nine times the throughput of atomic operations to global memory as well as adding support for additional atomic operation functions.
- The SMX processor contains 16 texture units, four times as many as previous generations. Kepler also removed the restriction of accessing only 128 textures simultaneously, allowing textures to be mapped at any time.
- *Dynamic Parallelism* effectively allows the GPU to keep itself occupied by providing means to launch its own additional kernel, creating streams and managing dependencies without needing to interact with the host. This functionality was also introduced into the programming model in CUDA 5.0 [93]. Dynamic parallelism allows for a greater variety of parallel algorithms to be implemented on the GPU.

At the time of this thesis publication the Maxwell architecture has become the latest version, the first cards being the GeForce GTX 750 and 750Ti as well as many of the GeForce 800M series cards incoporating this architecture, and more recently the GeForce 900- and 900M series, as well as a few Quadro cards. With many improvements on the Kepler architecture, most notably the Maxwell architecture increased the size of the L2 cache to 2MB (from 1536kB for the GK110 architecture), reducing the required memory bandwidth for non-coalesced memory access, while also supporting the new CUDA Compute Capability 5.0. Although this would be the optimum architecture to use, only a card using the Kepler architecture was able to be obtained.

## 3.3 CUDA Memory

Host and device have separate memory. In CUDA, memory can be stored in various ways and different types of memory work faster and perform more efficiently in certain situations [70].

GPUs contain a number of specific memory types which can be used explicitly by developers to cache memory access:

- *Global* memory is accessible from anywhere on the device - this means any thread running the program. It is the largest memory on the GPU and is *off-chip*, having the slowest access time of any memory type. Using *coalesced* memory transactions can improve the

performance of global memory but this is not always possible depending upon the application. Coalescing generally refers to when consecutive threads access consecutive memory addresses. When designing algorithms for use with the GPU, care must be taken to make sure memory is allocated efficiently for the best performance possible. Trade-offs between memory access complexity, number of memory accesses, and number of computations performed need to be taken into account because coalesced memory access is significantly faster than non-coalesced [3].

- *L1 and L2 caches*. The Fermi architecture introduced a single unifed memory request path for loads and stores - the *L1 cache* providing store and load operation service per SM multiprocessor, and the unified *L2 cache* providing service for all operations (store, load and texture) [94]. The size L1 cache can be configured as needed; the total on-chip memory is 64KB, which can be split between both the shared memory and the L1 cache. It may be divided into 48KB shared, 16KB L1 or 16KB shared, 48KB L1 on Fermi architectures, while Kepler architectures allow an additional split of 32KB of memory to both locations. The L1 cache allows for caching of local memory operations if using the standard Kepler architecture, however Tesla cards may also use the L1 cache for global memory operations. The L2 cache provides fast and efficient data sharing across the entire GPU. GK110 provides an L2 cache size of 1536KB (up from 768KB in Fermi architectures).

- The *read-only data cache* is a new method of utilizing the on-chip texture cache for read-only data. Originally this cache was accessible by mapping data as textures; the Kepler architecture modified the cache to be directly accessible to the stream multiprocessor. The size of the read-only data cache is 48KB [94].

- *Shared* memory is on-chip memory shared between all the threads on an SMX. It is relatively fast memory and can be used by threads in the same block to communicate with one another when needed. It is particularly advantageous when a thread block can load a block of data into on-chip shared memory, process it there, and then write the final result back out into external memory [70]. As mentioned previously, the total on-chip memory is split between shared memory and the *L1 cache*.

- *Texture* memory is a section of global memory which has been cached for efficient access. It is specifically designed for fast access to images used for texturing in computer graphics. Using coalesced global memory reads are the most efficient way to read device memory, but sometimes algorithms are unable to read memory using such a regular pattern. Texture memory is thus often used for data that is unable to be coalesced in global memory. It performs spatial caching and can potentially improve performance if consecutive reads access data at addresses which are within a one, two or three dimensional blocks of memory.*Texture fetching* is the process of reading a texture is typically done using the function `tex1Dfetch`.

- *Constant* memory is read-only memory, stored in global memory but also cached for efficient access. Constant variables are often used to provide input values to kernels.
- *Registers* are the fastest type of memory on the device and are used by single threads to store local variables. In the GK110 architecture, each thread has access to up to 255 registers.
- *Local memory* is any memory which is private to the thread - this includes the registers. Local memory is used once all other streaming multiprocessor resources have been used up, for example register spilling. It is stored in the L1 cache first, but if this cache is full it is moved to the L2 cache, and if it cannot be stored there, it is stored in global memory. Regardless of where it is stored, each thread still has access to its own local space.

While it is possible to achieve speed-ups of simulations using only global memory on the device, true performance optimization occurs through good management of all types of memory as each type has strengths and weaknesses. This thesis will attempt to show usage of most of the memory types to good effect in different simulations, although there is always room for improvement, especially as graphics cards are quickly becoming more and more powerful, with more options for programming them being added all the time as the CUDA programming model evolves.

## 3.4 Parallelization of Particle Methods

Parallelization of particle systems is by no means a new idea, and it existed even before the invention of GPU programming. Sims describes techniques used to animate and render particles with a data-parallel supercomputer [118], programmed in *Starlisp*, an extension of the Lisp programming language designed specifically for usage with parallel instructions and variables. More recently, Kolb et al. introduced a GPU implementation of a particle system using fragment shaders to render the dynamically-growing system [66].

The simplest way to parallelize particle systems is to divide the particles up between all threads executing in parallel. The functions governing the behaviour of each particle is executed as a CUDA kernel, allowing each particle to be represented by a single thread. This will be referred to as *per-particle parallelization*.

The memory access is highly dependent on the layout of the particles in memory. If particles are randomly distributed from 1 to $N$ (which is expected in a long running simulation) then the performance will suffer as memory is not coalesced. The solution to this problem is sorting the particle memory to improve data locality [3].

**Algorithm 1** the general layout of a particle system run on CPU

```
Given arrays P of n particles with position, velocity and force
Initialize all forces to 0
for all particles P do
  Let P1 be the current particle
  for all particles P do
    Let P2 be the second particle
    if P1 != P2 then
      Displacement d = P1.position - P2.position
      if d ≤ cutoff radius then
        newForce = calcForce(d)
        P1.force += newForce
      end if
    end if
  end for
  Integrate P.position and P.velocity using new P.force
end for
```

**Algorithm 2** the general layout of a particle system kernel on GPU

```
Given arrays P of n particles with position, velocity and force
Given thread index i
Initialize all forces to 0
Particle P1 = P[i]
for all particles P do
  Let P2 be the second particle
  if P1 != P2 then
    Displacement d = P1.position - P2.position
    if d ≤ cutoff radius then
      P1.force += calcForce(d)
    end if
  end if
end for
Integrate P.position and P.velocity using new P.force
```

To demonstrate the difference between a sequential and parallel implementation of a particle system, consider Algorithm 1. This algorithm might represent one iteration of standard particle simulation run on the CPU, where the program loops through each particle one by one, and then looping through all particles a second time to calculate the new force between each pair of particles. In contrast, a parallel implementation of the same code, shown in Algorithm 2 uses as an index into the particle array to find the current particle straight away, before looping through the other particles in the system and calculating particle pair forces, and then finally performing the numerical integration to calculate the position and velocity. The only additional code that is needed is setting up the indices for accessing the different parts of the data. Although in terms of parallel algorithms this is fairly inefficient (looping through the entire system could be terribly slow if the system is large), in contrast to the sequential version the parallel approach is more efficient.

## 3.5 CUDA Libraries

Two main CUDA libraries were utilized when developing the code for this thesis. Thrust [92], a library based of the C++ STL for implementing high performance parallel applications, is utilized for sorting operations. CURAND [90] is used for random number generation in parallel. An overview of both libraries are briefly covered in Sections 3.5.1 and 3.5.2.

### 3.5.1 Thrust

The Thrust template library [92] contains numerous high-performance parallel algorithms allowing programmers to optimize these parts of their code quickly and efficiently. Thrust is based on the C++ STL and is installed with the CUDA Toolkit. All that is required to use thrust functionality is including the relevant header files in the CUDA code.

More notable usages of Thrust when regarding parallel particle implementations include the `zip_iterator`, which allows multiple particle attributes to be packed together into a single vector and iterated over at the same time. For instance, during an iteration of numerical integration, the particle's position, velocity and force can be zipped and iterated over using just one iterator instead of three.

For parallel implementations of particle systems, organizing particle spatial data becomes very important. This will be covered in greater detail when describing the spatial grid approach in Chapter 5, but for now consider the basic problem that particles must be sorted with respect to their position in a three dimensional world space. One way of doing this is to divide the world space into a three dimensional grid and from the bottom-up link particles to particular cells in the grid. Sorting of these cells needs to be efficient as there will be many particles in a large system simulation and using inefficient sorting algorithms will slow down performance.

THRUST's `sort_by_key`, a *radix sort* sorting algorithm. Radix sort is one of the best known sorting algorithms and is very efficient for sorting *small keys*, assuming each key is represented as an integer number in some radix notation [114]. Radix sorts can fall under two main variants, *least significant* and *most significant* digits (LSD and MSD respectively). LSD is the better choice for sorting integer keys, while MSD is better suited for sorting characters [97]. LSD groups keys based on the least significant digit of each key, and sorting these digits usually with a bucket or counting sort [21]. This process is repeated for more significant digits. Radix sorts are often alternatives to comparison-based sorting algorithms such as the *merge sort*, which are generally more efficient for more complex keys.

Thrust also provides prefix-sum scan operations are also very useful for parallel sum calculations. For example, an exclusive prefix sum scan can be used for calculating the total number of cells in a grid which are occupied by particles. Exclusive scan is used in the generation of surfaces in Chapter 7.

```
curandState localState = state[id];

//Circular base
float theta = (2 * PI) * curand_uniform(&localState);
float r = curand_uniform(&localState);

pos[id].x = 0.2 * (r * cos(theta));
pos[id].y = 0.0f;
pos[id].z = 0.2 * (r * sin(theta));

//Make red more prominent towards the outside of the flame
col[id].x = 1.0f;
col[id].y = (0.5 + 0.3 * curand_uniform(&localState)) * (1-r);
col[id].z = (0.0 + 0.5 * curand_uniform(&localState)) * (1-r);
```

*Listing 3.2: Code fragment of a fire system kernel using CURAND randomization.*

### 3.5.2 CURAND Random Number Generation

The highly randomized structure of some simulations demands efficient generation of random numbers but this is not so simple when generating in parallel. If each particle has to randomize attribute values, the thread may not be able to guarantee that the random seed it is using is unique in the particle system, and this would have the effect of many particles generating the same random numbers and thus behaving in exactly the same way. Considering the size of particles in an average system is quite large, it is important to make sure every thread has a unique random seed of good quality. The CURAND library takes care of this.

CURAND [90] is a random number generation library designed specifically for use with CUDA. It allows generation of high quality pseudo-random and quasi-random number sequences efficiently in parallel. CURAND provides functionality for random number generation on both the host and device. If generated directly on the device, the random numbers are stored in global memory. In order to use the library, the header files `curand.h` and `curand_kernel.h` need to be included in the program. CURAND libraries are constantly being updated. The current version v6.5 comes with the CUDA 6 Toolkit. The simulation itself only needs to make use of the basic random number generation functionality and the libraries are capable of much more complex generation. The following is a brief explanation of how to use the library at a basic level in relation to the particle system.

An array of the special type `curandState` is initialized for use on the *device only* (it needs no host equivalent). Each particle in the simulation should have its own random number generator, so memory is allocated on the device for the number of particles in the simulation times the size of the `curandState` type. Before any random numbers can be generated, the random number generator must first be initialized to setup the CURAND states, using the function `curand_init`. This function will setup an initial state from the given `seed`, sequence number (`id`) and offset (`0`). The pseudo-random number sequences generated have a period of at least $2^{190}$ [90].

It is best practice to use a unique seed every time the random number generator is initialized, and multiple kernel launches should use the same seed but assign different sequence

| Number of Cores | 2304 |
|---|---|
| Clock Frequency (MHz) | 863 |
| Texture Fill Rate(billion/sec) | 160.5 |
| Memory Speed (Gbps) | 6.0 |
| Memory Configuration | 3072 MB GDDR5 |
| Memory Interface Width | 384-bit |
| Memory Bandwidth | 288.4 |

*Table 3.2: NVIDIA GeForce GTX 780 Specifications*

numbers in a monotonically increasing way [90]. This is sufficiently random as long as the random numbers do not have to be unpredictable; for example, this would not be sufficiently random for security-related problems such as password generation. However, for the purposes of random number generation for particle systems, this method will efficiently produce pseudo-random numbers of sufficient quality in parallel.

Once the CURAND states have been initialized, the main simulation kernel can be executed. A random number can be generated from a given state by calling variations of the `curand()` function. In this case, `curand_uniform` is used to generate a random number in a uniform distribution between *0.0* and *1.0*. Other distributions such as the normal and poisson distributions are also available, as well as double-precision variants of these. An example of using CURAND random number generation is shown in Listing 3.2, a portion of a global kernel call for updating particles which use `curand_uniform` to randomize a particle's initial starting position and colour variables.

## 3.6 Graphics Cards

The graphics card used both for the performance testing and the rendering of the simulations in this thesis is the NVIDIA GeForce GTX 780. It uses the Kepler GK110 architecture and Table 3.2 shows the main specifications for this card.

## 3.7 Summary

This chapter outlines the parallelization component of the thesis, introducing the CUDA programming model and describing the general method of parallelizing a particle system. CUDA allows particle systems to be parallelized effectively, and provides useful libraries such as THRUST and CURAND to perform common parallelizable tasks. Particle systems are generally parallelized on the particle level i.e. one thread per particle. Each of these threads will executed a set of instructions depending on which particle model is being used. In the next chapter, the models of the different particle systems used in this thesis will be explained, including both fire and water methods and why these models are suitable for real-time simulation.

# CHAPTER 4

# PARTICLE SYSTEM MODELS

There are a wide variety of approaches to modelling a particle system. The type of model chosen depends on a number of factors:

- *Complexity* of the desired behaviour of the system. Certain particle models are more complicated than others, requiring more simulation steps, greater memory requirements and more advanced algorithms for calculating particle forces and velocities. This is particularly relevant to real-time simulations, as the particle method complexity becomes restricted to methods that can be run in real-time. Parallelization of these methods helps with this immensely.
- *Behavioural* aspects of each mode. Certain particle models work better in modelling certain behaviours. For example, fluid-based particle models such as Smoothed Particle Hydrodynamics work particularly well for systems intending on simulating water or other viscous fluids, while hard-body collision type approaches are poorly suited to the task because they provide few tensive forces keeping the particles together as a fluid.
- *Rendering* factors need to be taken into account if the type of rendering method incorporates properties of particles not present in simpler particle methods. For example, using particle density as a means of colour calculation can only be performed on a particle method which has density as a parameter.

This chapter will describe some different particle system modelling approaches, with varying degrees of complexity and inclined towards different behavioural aspects. Section 4.1 will describe a non-interacting particle system, the simplest of the ones featured, involving no particle interaction. Section 4.2 describes the spring-mass model, or a soft-body dynamics model, where, as the name suggests, particles are connected with conceptual springs, exerting forces on each other along these springs. Section 4.3 will describe the Smoothed Particle Hydrodynamics model, a computational fluid dynamics model based on particle velocity and pressure forces. Section 4.4 will describe the velocity-vortex model, another fluid dynamics approach that uses the *curl of the velocity* (the vorticity) to update the particle velocity.

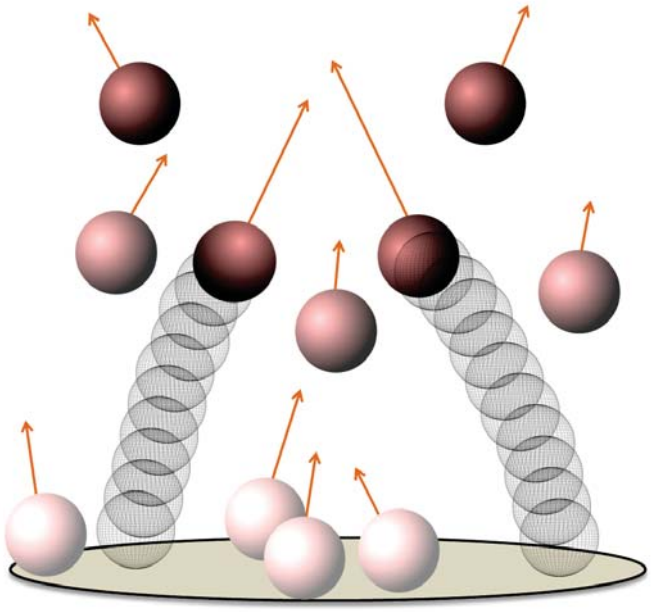

*Figure 4.1: An example of non-interacting particle motion, where particles are created in a circular area at the bottom of the simulation space, and are able to move freely in any direction. There are no interactions between pairs of particles. This particular example is in relation to a simple fire particle system, where particles generally move upwards, with varying x- and z-trajectories.*

## 4.1 Non-interacting Particle Model

A *non-interacting* particle model is a term used to describe a particle system in which particles are unable to interact with each other in any way. Interactions would be considered as any time two particles have an effect on each other. This could be physically colliding with each other or attracting or repelling each other from a distance.

Central to the non-interacting particle system model is the idea of *creation and destruction*. Particles are created by some initialization function, initializing the particle emitter position and other variables depending on the type of system that is being simulated. The particle is then set to a timer, whereby once the timer runs out, the particle is destroyed and recreated(optionally) at another position. Recreation is optional because, depending on the type of phenomena intending on being simulated, particles may not need to be preserved throughout the entirety of the simulation. For example, the usage of particles to simulate explosions - the explosion should only happen once, so particles must be created at the beginning of the explosion and destroyed at the end. On the other hand, particles simulating a fire, or a river running through a scene, would have particles reposition at the bottom of the fire, or top of the river, once they reach the extreme end of their flow. Figure 4.1 shows the general idea of a non-interacting particle system.

Thus several additional attributes must be added to each particle to allow for this creation and destruction process to occur. The *current life* of a particle simply refers to the accumulation

of time since a particle has been created. In the simulation, the maximum life of each particle is randomized upon initialization and the current life begins at this value. In each iteration of the simulation loop, the timestep is subtracted from the life value, multiplied by some *decay rate*. Thus the life value is actually an indicator of the time *remaining* before the particle is destroyed. The *decay rate* is also randomized upon initialization, and as such certain particles will lose life faster than others. When the particle's life count reaches zero or below, it is destroyed and no further calculations can be performed on it until the following loop iteration where it will be recreated with new random values.

Because of the simplicity of the non-interacting particle models, they are fairly easy to implement and run extremely fast. An implementation of this particle model will be demonstrated in Chapter 6. However, these types of particle models suffer in trying to simulate highly complex behaviour, and often have to resort to high usage of randomization to give the appearance of chaotic motion. Other, more physics-oriented approaches are more suited to these types of problems.

## 4.2 Spring-Mass Model

Spring-mass models are most often used in modelling cloth-related simulations [108] [32] [7], but these models are suited to many other applications as well, including surface modelling [124] and crowd simulation [111].

As mentioned previously, for particle systems where soft-body dynamics are observed a spring-mass model is applicable. The interactions between particles can change depending on the types of springs used, as well as the distances they are compressed or extended.

In a spring-mass particle model particles are modeled as points connected by conceptual springs, which are weightless and serve simply as a vector between any two particles along which an *elastic force* is calculated. Springs obey the laws of physics according to *Hooke's Law* [113], which states that the force required to compress or extend a spring is dependent on the compression or extension distance, and the *stiffness* of the spring itself. The most general form of the law is shown in Equation 4.1.

$$\vec{F} = -k_s\vec{X} \tag{4.1}$$

Here the force $F$ is directly proportional to the compression or extension distance $\vec{X}$, multiplied by the spring stiffness constant $k_s$. Additionally, the reaction force of the spring should be equal to the applied force, following Newton's third law, so $\vec{F}$ is therefore directly proportional to the compression or extension distance in magnitude, but acting in the opposite direction $-\vec{X}$. For example while being compressed together, at any instance of time there should be an equal and opposite force extending the spring. Figure 4.2 shows this general idea. Notice that in this example, the spring mass system is in relation to a cloth simulation rather than a

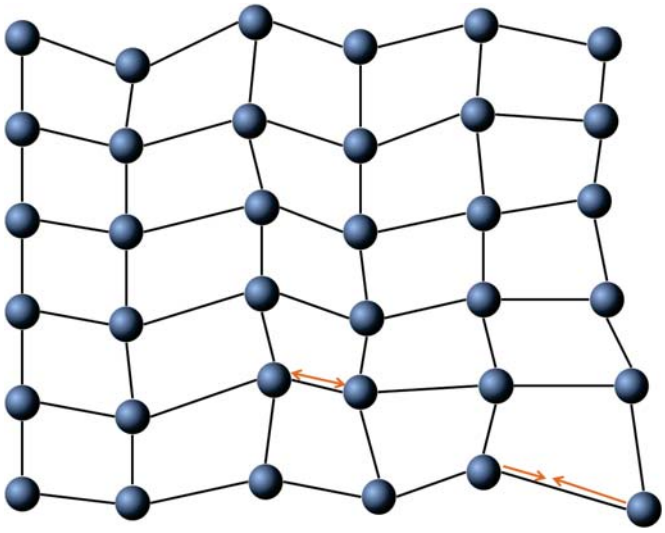

*Figure 4.2: An example of spring-mass particle motion. Depending on the compression or extension distance of the springs connecting pairs of particles, a force is exerted on each particle. If the spring is compressed, the pair pushes each other apart, while if the spring is extended, the pair are drawn together.*

water simulation. A water simulation-specific approach to the spring-mass model is discussed in Chapter 6.

When applying Hooke's law to the particle system, the spring rest distance $r$ should be determined. Particles with a scalar distance equal to the rest distance are situated at the equilibrium point where the conceptual spring is neither being compressed nor extended. A distance larger than the rest distance implies the spring is being extended, while a distance shorter than the rest distance implies the spring is being compressed. This application is shown in Equation 4.2, where $|X_a - X_b|$ is the scalar distance between the two particles $a$ and $b$, while $X_{ab}$ is the relative position of the two particles (the spring vector).

$$\vec{F} = -(k_s(|\vec{X}_{ab}| - r)\frac{\vec{X}_{ab}}{|\vec{X}_{ab}|} \tag{4.2}$$

In its simplest form, this equation gives the value of the *spring force* of the spring-mass system, but this is just one part of the total force exerted by one particle on another. Particles experience *friction* or *drag forces* when moving, depending on the velocity of the particle as well as a *drag constant* as shown in Equation 4.3. Friction or drag forces are approximated by *damping*.

$$\vec{F}_d = -k_d\vec{v} \tag{4.3}$$

where $\vec{v}$ is the velocity of the particle and $k_d$ is the drag constant. Particles can be assumed to

be in an environment without friction if the drag constant is zero.

Particles also experience *shear forces*, forces which occur when two parts of a body move in different directions. Basically, the shear force is the equivalent of a spring force which does not align with the spring vector. Shear forces apply to the tangential component of the relative velocity of the two particles, also having a shear coefficient property of the system determining the magnitude of the shear force. This is shown in Equation 4.4.

$$\vec{F}_{sh} = k_{sh}(\vec{V}t_a - \vec{V}t_b) \tag{4.4}$$

where $\vec{V}t$ is the relative tangential velocity between the two particles, and $k_{sh}$ is the shear constant.

Finally, particles may exert *attraction forces* on each other. The type of attraction force depends on the type of system that is intending to be simulated. In an *n-body* simulation [105], the attraction force of one particle on another refers to the gravitational pull of the particle based on its mass. This gravitational force is shown in Equation 4.5.

$$\vec{F}_g = \frac{Gm_a m_b \vec{X}_{ab}}{|\vec{X}_{ab}|^3} \tag{4.5}$$

where $m_a$ and $m_b$ are the masses of the two particles, $\vec{X}_{ab}$ the relative position of the two particles, $|X_{ab}|$ the scalar distance between the two particles and $G$ is the *gravitational constant*, approximately $6.674 \times 10^{-11} Nm^2 kg^{-2}$.

The superposition of all these forces make up the *total internal force* of the spring-mass particle system. When applying these forces to the particle system, the total force $\vec{F}_T$ acting on any particle $a$ by any other particle $b$ is shown by in Equation 4.6.

$$\vec{F}_T = \vec{F}_s + \vec{F}_d + \vec{F}_{sh} + \vec{F}_g \tag{4.6}$$

Perhaps the most important aspect of the spring-mass force calculation is setting the spring rest distance $r$. For a cloth simulation or n-body simulation, the rest distance might be large in comparison to the size of the particle. However, for soft-body collision dynamics this need not be the case. Setting the rest distance to the sum of the radii of the two particles means the spring becomes compressed only when the two particles touch each other and thus produce a repulsive force keeping particles at least the sum of their radii away from each other (a soft-body collision).

Additionally, spring forces need not always be applied for both extension and compression. Specifying the spring force *only* to act when particle distances are less than $r$ and not greater means that there will be no reactive compression forces on the particles as they move away from

**Algorithm 3** Application of the spring-mass model in parallel

```
Given array X of n particle positions
Given array V of n particle velocities
Given particle thread index i
Local environment cell size 2r
//Set up local environment
Calculate neighbouring cells C
//Loop through the local environment, calculate the internal force
for each particle P at X[i] do
  Calculate local environment based on P
  for all 27 (max) neighbouring cells C do
    Calculate first f and last l ordered particles within cell
    //Avoid case where particle interacts with itself
    for j = f; j ≤ l; j++ do
      if i ≠ j then
        Perform collision detection for P and particle at X[j]
        Calculate internal force F, based on X and V
      end if
    end for
  end for
  Add external force E where applicable
end for
```

each other - this is the approach used later in Chapter 6 for simulating a crude fluid simulation using this model.

Algorithm 3 shows the approach used to apply the model in parallel to a particle system. Since each particle only interacts with other particles within its local environment, the algorithm assumes the setup of this local environment before the forces are calculated for each particle. This local environment typically consists of a grid of cells surrounding the particle, with dimensions $3 \times 3 \times 3$ to make 27 cells altogether. Setting up this grid of cells is covered in more detail in Chapter 5. It is assumed that particles within each cell in the grid are sorted such that they are accessible efficiently in a parallel manner. The size of the local environment cells is typically dependent on the rest distance $r$.

## 4.3 Smoothed Particle Hydrodynamics

*Smoothed Particle Hydrodynamics* (SPH) is a computational method first proposed by Gingold and Monaghan [38] and Lucy [76] for simulating fluids. Originally it was designed for use with astrophysics, but since has been utilized extensively in simulation of fluid flows [125] [20] [27] [57], as well as self-gravity effects [9], lava flows [121] and deformable bodies [30]. SPH was also used to model fire and other gaseous substances by Stam and Fiume [120]. More recently, implicit approaches to SPH [57] [22] have also been developed.

SPH is a velocity-pressure particle model and is relatively more complicated than the particle models previously described. The characterizing aspect of SPH is the use of a *smoothing*

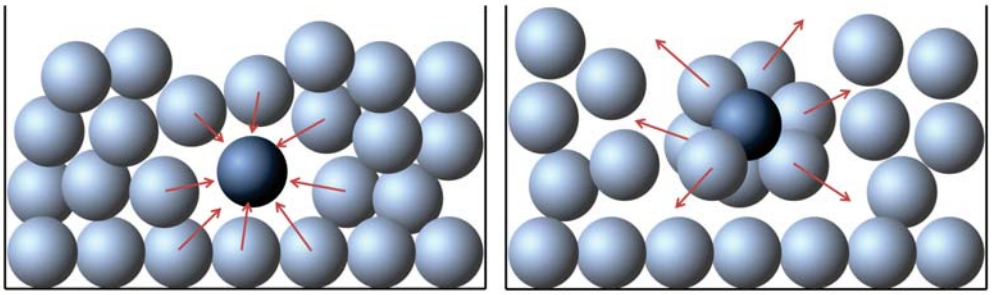

*Figure 4.3: SPH is a pressure-velocity particle model. As shown in the left-hand diagram above, particles move towards low pressure areas in the fluid. In contrast, high pressure areas push particles away from each other, as shown in the right-hand diagram.*

*kernel* to smooth the properties of the elements in the fluid. This is done by sampling element properties from other surrounding elements within the fluid. Smoothing consists of performing a simple interpolation across elements or particles within a certain field size to obtain approximated values for field quantities such as pressure or density. Figure 4.3 shows the basic idea of SPH.

To describe the smoothed particle hydrodynamics method properly, it is first necessary to describe the basics in fluid dynamics as a whole. Firstly a quick overview of computational fluid dynamics theory is provided, and then afterwards a general overview of the SPH algorithm and governing hydrodynamics equations. This include the most commonly used smoothing kernel functions.

### 4.3.1 Navier-Stokes Fluid Flow

Computational Fluid Dynamics (CFD) was explained in Section 2.2. CFD approximates the behaviour of fluids by using numerical methods. CFD seeks to solve the Navier-Stokes equations [48] [129] [19] describing the motion of fluid. As SPH is a Lagrangian (particle-based) method, the Lagrangian form of the Navier-Stokes equations for incompressible fluid flow should be used and is shown in Equation 4.7.

$$\frac{\delta \vec{v}}{\delta t} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \vec{v} + g \tag{4.7}$$

where $\mu/\rho$ is the *kinematic viscosity* of the fluid, which is not to be confused with the *dynamic viscosity* $\mu$. $\rho$ is the density of the fluid, $p$ the pressure of the fluid and $g$ is the acceleration due to gravity.

To simplify the equation, the total acceleration is divided up into three distinct parts. The acceleration due to *pressure* given by $-(1/\rho)\nabla p$, moves the fluid particles from high-pressure areas to low pressure areas. The acceleration due to *viscosity* given by $(\mu/\rho)\nabla^2\vec{v}$, keeps the fluid together when the velocity of particles applies large shear forces. Finally the *external* acceleration is usually represented by $g$ (acceleration due to gravity).

The SPH equations approximate the Navier-Stokes equations by representing the volume of the fluid as a set of discrete points (the particles) and interpolating over these to calculate the fluid properties [86]. These equations are described in detail in the next section.

### 4.3.2 SPH Equations

The general equations for SPH have been published many times in revisions by Monaghan [86] [84] [85] as well as later applications of SPH [87] [30]. The general equation can be expressed for *any* given quantity $A$ (for example density) at some point $r$ and using some smoothing kernel function $W$. Equation 4.8 shows this general equation.

$$A(r) = \sum_b m_b \frac{A_b}{\rho_b} W(r - r_b, h) \tag{4.8}$$

where $\rho_b$ is the density of particle $b$, and $h$ refers to the *half-width* of the smoothing kernel, also known as the *smoothing length*.

The larger the size of $h$, the "smoother" the kernel. This inevitably means that more neighbouring particles will need to be interpolated and thus will be more expensive to compute. There are several possible smoothing kernel functions, which will be covered later in the section.

So to calculate the properties of a particle $i$ in the fluid, the general equation is shown in Equation 4.9. Derivatives to the general function can be found using the smoothing kernel derivatives, which is also shown in the equation.

$$\begin{aligned} A_i &= \sum_j m_j \frac{A_j}{\rho_j} W(r_i - r_j, h) \\ \nabla A_i &= \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r_i - r_j, h) \\ \nabla^2 A_i &= \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(r_i - r_j, h) \end{aligned} \tag{4.9}$$

Given Equation 4.8, the density of the particle $i$ can then be written in terms of the properties of particle $j$ as shown in Equation 4.10, as well as the simplified form.

$$\begin{aligned} \rho_i &= \sum_j m_j \frac{\rho_j}{\rho_j} W(r_i - r_j, h) \\ \rho_i &= \sum_j m_j W(r_i - r_j, h) \end{aligned} \tag{4.10}$$

The pressure $p$ can be calculated using the *law of ideal gas* [87], stating the pressure of a fluid is dependent on the volume of the fluid per unit of mass, the temperature of the fluid, and the universal gas constant. Assuming the fluid maintains a constant temperature (an isothermal

fluid), the pressure of the fluid can be simplified in terms of the *gas stiffness constant* $k$ as shown in Equation 4.11 as a function of the mass-density $\rho$ and the rest-density $\rho_0$ [30].

$$p = k(\rho - \rho_0) \tag{4.11}$$

$$f_{p_i} = -\frac{1}{\rho}\nabla p = -\sum_{j \neq i} m_j \left( \frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(r_i - r_j, h) \tag{4.12}$$

Applying the pressure element of the Navier-Stokes equation for incompressible fluid (Equation 4.7) and the general SPH equations produces the equation for pressure force. Because the forces between two particles must be conserved (Newton's Third Law) Equation 4.12 applies a symmetric approximation [84] to the force. It conserves momentum of the system by providing a symmetrical pressure force that would otherwise not occur if the general equation is applied. Pressure forces exerted on one particle would otherwise be solely dependent on the pressure of the other particle, which will inevitably be different for both particles and thus conservation of momentum will not occur.

Finally, viscosity is calculated by combining the viscosity element of the Navier-Stokes equation (Equation 4.7) and the general SPH equation, resulting in Equation 4.13. Viscosity determines the strength of the fluid as it is moving - a large velocity inevitably causes the particles to break apart, but adding a viscosity force prevents this to a certain degree. The viscosity constant $\mu$ is used to determine the viscous strength of the particles - higher viscosity will make the particles stay together at more turbulent velocities.

$$f_{vi} = \mu \nabla^2 \vec{v} = \mu \sum_{j \neq i} m_j \frac{(\vec{v}_j - \vec{v}_i)}{\rho_j} \nabla^2 W(r_i - r_j, h) \tag{4.13}$$

Several different smoothing kernel functions have been suggested by various sources [38] [76] [55] [87]. Smoothing kernels use the difference in positions of the two particles as well as the smoothing length to calculate the different effects required in SPH. Care needs to be taken when choosing different smoothing kernels as higher-order kernels can be computationally expensive despite producing more accurate results.

A set of smoothing kernels suggested by Müller et al [87] are used in this implementation to produce a stable simulation as well as being relatively inexpensive to compute. Because the coefficients of the smoothing kernels do not require the calculation of $r$, the smoothing kernel coefficient is based solely on the smoothing length $h$ which is constant. It makes sense then to pre-calculate these coefficients at the beginning of the simulation and store them in constant memory for easy access later.

The *poly6* smoothing kernel shown in Equation 4.14 is used to calculate the particle cell density. For pressure, the *spiky* kernel [30] is used, also shown in Equation 4.14, and finally, the viscosity smoothing kernel, also given in the equation.

$$
\begin{aligned}
W_{poly6}(r,h) &= \frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3 & 0 \le r \le h. \\ 0 & \text{otherwise}\end{cases} \\
W_{spiky}(r,h) &= \frac{15}{\pi h^6}\begin{cases}(h-r)^3 & 0 \le |r| \le h. \\ 0 & \text{otherwise}\end{cases} \\
W_{viscosity}(r,h) &= \frac{15}{2\pi h^3}\begin{cases}-\frac{r^3}{2h^3}+\frac{r^2}{h^2}+\frac{h}{2r}-1 & 0 \le r \le h. \\ 0 & \text{otherwise}\end{cases}
\end{aligned}
\tag{4.14}
$$

When calculating the pressure force, the first order differential of the spiky kernel [30] is used - the first-order differential is required because of Equation 4.12. The first order differential is shown in Equation 4.15. Similarly, because of Equation 4.13, the second-order differential (laplacian) of the viscosity kernel is required when calculating the force due to viscosity. This differential is also shown in Equation 4.15.

$$
\begin{aligned}
\nabla W_{spiky}(r,h) &= -\frac{45}{\pi h^6}\frac{r}{|r|}\begin{cases}(h-r)^2 & 0 \le r \le h. \\ 0 & \text{otherwise}\end{cases} \\
\nabla^2 W_{viscosity}(r,h) &= \frac{45}{\pi h^6}\begin{cases}h-r & 0 \le r \le h. \\ 0 & \text{otherwise}\end{cases}
\end{aligned}
\tag{4.15}
$$

The SPH can be applied to the particle system as follows in Algorithm 4. Similar to the spring-mass model, particles only interact within their own local environment. In the case of SPH, their local environment is defined by the size of the smoothing kernel. Like the spring-mass model, it is assumed the particles within the local environment are sorted in an order suitable for efficient parallel access - for more detail on this, see Chapter 5.

The SPH simulation requires an additional pass through the particle's local environment to initially calculate the new densities and pressures before they can be used for calculating the pressure forces. This increases the execution time of the simulation by a relatively large amount depending on the size of the particle system, but as always this is a trade-off to the increased realism and physical correct-ness of the simulation.

## 4.4 Velocity-Vortex Model

The velocity-vortex model is an additional model used in this thesis to explore more complex methods of simulating fire systems, just the SPH was used as a comparison to a simpler spring-mass model for the simulating of water. It was first proposed by Leonard [71] for two-dimensional fluid flow simulation, and then in three dimensions by Novikov [89]. Novikov defines the term *vorton* as a three-dimensional vortical singularity [89], which are formed in fluid dynamics when vortex filaments are stretched. Because it is difficult to predict the for-

**Algorithm 4** Application of SPH in parallel

```
Given array X of n particle positions
Given particle thread index i
Given array V of n particle velocities
Given array D of n particle densities
Given array Pr of n particle pressures
Given local environment cell size h
Calculate smoothing kernels W based on h
//Set up local environment
Calculate neighbouring cells C
//Perform a first pass, calculating density and pressure for each particle
for each particle P at X[i] do
  Calculate local environment based on P
  for all 27 (max) neighbouring cells C do
    Calculate first f and last l ordered particles within cell
    for j = f; j ≤ l; j++ do
      Accumulate density into D[i] from each particle, smoothed with W(density)
    end for
  end for
  Calculate the pressure Pr of particle P, based on the total density
end for
//Perform a second pass, using the total densities and pressures to calculate the force
for each particle P at X[i] do
  Calculate local environment based on P
  for all 27 (max) neighbouring cells C do
    Calculate first f and last l ordered particles within cell
    for j = f; j ≤ l; j++ do
      if i ≠ j then
        Calculate pressure force based on D and Pr, smoothed with W(pressure)
        Add viscosity force based on V, smoothed with W(viscosity)
      end if
    end for
  end for
  Add external force E where applicable
end for
```

mation of these singularities within the fluid flow, the singularities are instead introduced as an element of the fluid and from here fluid dynamics equations can be used to govern the fluid behaviour based on these singularities. The singularities or vortons are the equivalent to the particles in a particle system, simply with an added vortex element to them.

For vortex particles or vortons, a vortex "swirls" around the particle, inducing velocity in the surrounding fluid and thus the other surrounding particles. The description of how these vortex particles rotate in the fluid is known as the particle's *vorticity*. Cottet and Koumoutsakos define vorticity as a solid-body-like motion that is imparted to the elements of a fluid due to stress distribution [23]. Vorticity specifically refers to the *curl* of the velocity and is a direct measure to the particle's *angular velocity* $\vec{w}$, given by the relation $\vec{\omega} = \vec{w}/2$. As such, the

vorticity field equation is given simply by the change in velocity, as shown in the general equation 4.16 [23]. As vorticity refers to the *curl* of the velocity, it is always moving in a circular motion around the vorton.

$$\vec{\omega} = \nabla \times \vec{u} \tag{4.16}$$

where $\vec{\omega}$ is the vorticity field and $\vec{u}$ is the velocity field. This is not to be confused with the particle velocity $\vec{v}$, which follows the field velocity $\vec{u}$ using an average of the field velocity flows within the local area of the particle. The equation also means that the vorticity field is solenoidal ($\nabla \cdot \vec{\omega} = 0$), meaning that the vorticity of a field has zero divergence.

Discrete Vortex Methods (DVMs) are simulation methods which use these vortices as fluid elements, while discretizing the otherwise continuous fluid dynamics equations [40]. As was described in the Smooth Particle Hydrodynamics model, fluid dynamics models seek to solve the the Navier-Stokes equations of motion, and vortex methods are no different. It differs in that the velocity-vortex method solves for vorticity, rather than momentum. The two forms of the Navier-Stokes equations deal with fluid differently - the Eulerian form describes flow from a fixed point (the grid cell), while the Lagrangian form describes the flow from a fluid element (the particle) [23]. Taking the standard Navier-Stokes equation as described in Equation 4.7, the equation for vorticity can be derived.

Conservation of mass is given by the following Equation 4.17, expressed as a function of the velocity $\vec{u}$ [23].

$$\frac{\delta \rho}{\delta t} + \nabla \cdot \rho \vec{u} = 0 \tag{4.17}$$

The equation simply states that in an Eulerian view, the rate at which the mass density accumulates ($\frac{\delta \rho}{\delta t}$) in each cell should be equal to the net flow rate ($\nabla \cdot \rho \vec{u}$) out of the cell. In the Lagrangian view, the mass density of the particle is equal to the mass per unit of volume multiplied by the expansion rate.

Conservation of momentum is also expressed in terms of the velocity $\vec{u}$ and the pressure flow field $p$. When looking at this equation from an Eulerian view, it describes the rate of change in momentum in a cell, while the Lagrangian view would describe the rate of change in momentum of the particle.

$$\rho(\frac{\delta \vec{u}}{\delta t} + \vec{u} \cdot \nabla \vec{u}) = -\nabla p + \mu \nabla^2 \vec{u} \tag{4.18}$$

The rate of change in vorticity then is given using a similar equation to 4.18, shown in Equation 4.19, expressing the change in vorticity in terms of the velocity $\vec{u}$ and the viscosity $\mu$ [23].

$$\frac{\delta\vec{\omega}}{\delta t} + \vec{u} \cdot \nabla\vec{\omega} = (\vec{\omega} \cdot \nabla)\vec{u} + \frac{\mu}{\rho}\nabla^2\vec{\omega} \tag{4.19}$$

Equation 4.18 specifically refers to the Eulerian form of the Navier-Stokes equation when solving for vorticity - the Lagrangian is almost identical, and has been omitted simply for brevity. What should be understood about this equation is that the change in vorticity can essentially be broken down into two main terms - the *stress term*, given by $(\omega \cdot \nabla)\vec{u}$, and the *viscous term*, given by $\mu\nabla^2\vec{\omega}$. The stress term describes the change in vorticity due to velocity, while the viscous term describes the change in vorticity due to viscosity, or simply the *diffusion* or spread of vorticity through the fluid.

Utilizing aspects of both Eulerian and Lagrangian flows is used for the fire simulation using vortex methods. The reason for this is that particles in the fluid should *follow the flow* of the fluid. Therefore, some of the main steps of the vortex model is the calculation of the velocity of the fluid inside each cell in a *velocity grid*. As will be explained later, using a grid becomes convenient when methods improving spatial interaction between particles are already implemented (see Chapter 5). In any case, the velocity in each grid cell must be calculated based on the vortons inside the cell and neighbouring cells. The *velocity due to vorticity* is basically the advection of a particle's vorticity applied to the entire velocity cell. Figure 4.4 demonstrates this.

The velocity from vorticity can be given using *Biot-Savart's* law as shown in Equation 4.20, where $\vec{u}$ is the velocity of the grid cell, $\vec{\omega}$ is the vorticity of the vorton, $\vec{r}$ is the *vector distance* between the grid cell and the vorton, and *r* is the magnitude of this distance. When calculating vector distances between grid cell and vorton, it is common to use the bottom left-hand front corner as the reference point for the position of the cell.

$$\delta\vec{u} = \frac{\vec{\omega} \times \vec{r}}{4\pi r^3} \tag{4.20}$$

Once the velocity of the grid cell has been calculated, it is used in the standard integration method for all vortons residing in this particular cell. To do this, for each vorton in the simulation, the corresponding velocity is found using an interpolation function from the velocities of the surrounding grid cells, and is then applied to the vorton's position.

The *stress term* of the vorticity equation is often referred to as the *stretching and tilting* of the vorticity [40] and is another essential part of the simulation. Expanding the stress term results in a matrix of all partial derivatives of the velocity - i.e. the *Jacobian* matrix of the velocity. The equation for the Jacobian is shown in greater detail in Equation 4.21. Applying the Jacobian of the velocity at each velocity cell to all corresponding vortons inside that cell gives the stretching and tilting of the vorticity due to stress.

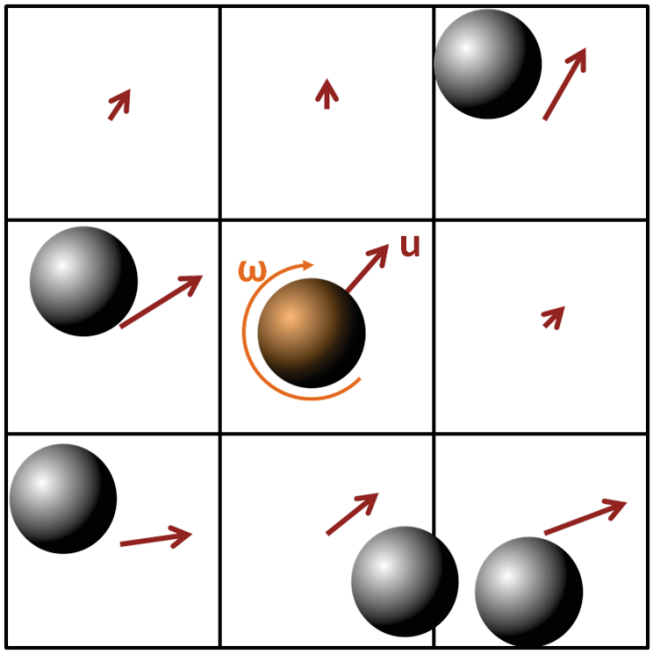


*Figure 4.4: The calculation of a grid cell's velocity based on the vorticity of the particles inside them. The velocity (red) is calculated for each cell in the grid depending on the vorticity (orange) from the particles inside. Even though there may be no particles in the cell, particles in neighbouring cells also apply a velocity field with vorticity, although this is relatively weaker than if the particle were directly inside the cell.*

$$(\vec{\omega}\cdot\vec{\nabla})\vec{u} = \omega_x\frac{\delta\vec{u}}{\delta x} + \omega_y\frac{\delta\vec{u}}{\delta y} + \omega_z\frac{\delta\vec{u}}{\delta z} = \begin{bmatrix} \frac{\delta u}{\delta x} & \frac{\delta u}{\delta y} & \frac{\delta u}{\delta z} \\ \frac{\delta v}{\delta x} & \frac{\delta v}{\delta y} & \frac{\delta v}{\delta z} \\ \frac{\delta w}{\delta x} & \frac{\delta w}{\delta y} & \frac{\delta w}{\delta z} \end{bmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{4.21}$$

The viscous term of the vorticity equation applies to what is called *vorticity diffusion* - i.e. how vorticity is spread between vortons throughout the system. Basically, between any two vortons, vorticity is exchanged based on the viscosity of the fluid as a whole as well as the distance between the two vortons. A low viscosity value means that very little vorticity is traded and therefore vortons are allowed to move pretty much where they want, rather than try and follow the other vortons (keeping the fluid together). High viscosity means that the vorticity of neighbouring vortons will influence each other greatly and will be far more inclined to move and stay together in the fluid.

There are many different methods of vorticity diffusion, including reducing vorticity artificially (damping) or exchanging it between vortons. The method used in this thesis is an exchange method known as *Particle Strength Exchange* (PSE) [140]. This method performs well in situations when fluid properties such as viscosity and density are already known. The equation for PSE is shown in Equation 4.22, given in terms of the viscosity $\mu$, the vorton radius $r$ and density $\rho$, and the vorticity of the vortons $\vec{\omega}$. It also employs a cutoff function $f$ using the

positions $\vec{x}_i$ and $\vec{x}_j$ of the two vortons. Note that the equation refers to the vorticity diffusion to any arbitrary vorton $i$, and the vorticity is exchanged throughout $N$ vortons in the current and neighbouring cells, not every vorton in the system.

$$\frac{\delta\vec{\omega}_i}{\delta t} = \frac{\mu}{\rho r^2}\sum_{j=1}^{N}(\vec{\omega}_j - \vec{\omega}_i)f(\vec{x}_j - \vec{x}_i) \tag{4.22}$$

The amount of vorticity traded is also dependent on the distance between the two vortons. Closer vortons diffuse more vorticity between each other when compared to two vortons far away from each other. This can further alleviate parallelization problems when it is considered that the vorticity diffusion between two vortons becomes almost zero as the distance becomes larger and larger, therefore it is possible to apply a cutoff point where neighbouring vortons need not apply diffusion to each other after a certain distance. This will be explained in greater detail in Chapter 6.

In addition to the velocity-vortex method being used to simulate fire to a greater degree of realism, an additional component to the simulation is the use of a rigid-body interaction with the vortons. Previously, the spring-mass model implements soft-body collision physics to interactions between particles, but rigid-body dynamics work slightly differently.

### 4.4.1 Rigid-Body Interaction

Rigid-body interaction is a deceptively broad topic and can be related to many different types of particle systems, in particular fluid simulations. A *rigid-body* is something which cannot be deformed - this is in contrast to a *deformable* body which is allowed to change shape or volume. Rigid-bodies have many of the same properties as a normal particle has - position, velocity, acceleration, size and so on - but unlike a particle it is not necessarily treated as a part of a much larger system, rather, it is its own entity. The interaction between rigid bodies and particle systems is a very interesting subject and has seen a great deal of research [25]. For example, Harada et. al. [46] describes methods of simulating rigid bodies coupled with fluid dynamics systems, while [44] shows how a rigid-body can be approximated by a number of joined particles to make the interaction between particle systems and more complex-shaped rigid bodies easier and more efficient, while Macklin et. al [78] describe a unified particle dynamics framework which includes the representation of rigid bodies using particles. Indeed, simulating a rigid-body on its own is very simple - the interaction is the difficult part, because the simulation needs to handle computations of interactions from every particle in the system with every rigid-body in the system. Luckily, the use of the spatial grid can once again help with this problem and since the rigid-body is able to use the same spatial grid indexing functionality that particles use, it can be treated the same way.

Rigid-body interaction has been brought up in this instance because the fire system (at least, a more complex one) technically requires at least one rigid-body interaction to take place. Recall in Section 6.2 that the non-interacting particle model for fire creates particles at a specific

point or within a specific area - the *emitter*. In more practical terms, the emitter can be considered the *fuel* source, where the fire burns off. Rather than having a simple circle as the emission area, it makes much more sense realistically for the fire to be burning from some particular object which it actually interacts with. For this reason, rigid-body interaction is one of the more central ideas to this more advanced fire simulation. Additionally, because the fire particles should never push within the fuel at all, rigid-body interactions are preferable in this case to soft-body interactions such as those used in the spring-mass model.

Basic rigid-body motion follows mostly the same physical laws as soft-body particles - for example, the standard kinematic equations as in Equation 2.5 as well as Newton's Second Law $F = ma$ all apply if the body is *not static* in the simulation. This is a point of note because the rigid-body does not always need to be moveable - indeed, this will not be the case for the fire simulation. However it is important to note that bodies may be influenced by the particle system interactions just as particles can be affected by the rigid-body. In the case of the fire, it is simply not required as the body fuel source does not move regardless of the movement of the particles around it. It would be an interesting idea, however, if the movement of the particles over the body caused some sort of destruction effect on the body itself - literally burning away the fuel - and seeing how this would effect the behaviour of the fire as the simulation progresses. Rigid-body interaction is important in the fire simulation because it induces a vorticity in the particles as they rise off the body.

Interacting with rigid bodies can be problematic when taking into account the discretization process. If a vorton is moved naturally inside the area of a rigid-body, it is pushed out and a vorticity is applied to continue it on. This is a simple approach and not very accurate, so it can cause some instability in the system because the position of the vorton is basically overwritten without any account for conservation of momentum. Consequently, this needs to be treated with care, but in practice the method works at least to obtain good visual results on the macro level.

## 4.5 Summary

In this chapter, the models for the different particle systems used in this thesis are explored, including the governing equations of the models as well as discussing the potential strengths and weaknesses for using them. For the fire simulation, the models described include the non-interacting particle model and and velocity-vortex model, while for the water simulation the models described include the spring-mass model and Smoothed Particle Hydrodynamics. The computational implementation of these models (using CUDA) will be discussed later in Chapter 6 and the visual implementation (using OpenGL) will be discussed in Chapter 7. Before this, however, it is important to discuss the elements of the implementation which persist over all the different models - specifically, the implementation for the spatial grid (not applicable for the non-interacting method) and the integration method. This will be the subject of the next chapter.

# CHAPTER 5

# SPATIAL GRID AND INTEGRATION METHODS

Processing interactions between pairs of particles, or in fact any simulated object, is a complicated task in physically-based simulations due to the complexity and scale of the simulation systems involved. Interaction between objects must not only be highly efficient, but the interactions which are produced must maintain a relatively correct physical response [126]. A *spatial grid* is designed to do this, whereby the simulation space is divided up into a grid of cells and interactions are processed based on the objects location within the grid. The spatial grid implementation will be covered in Section 5.1.

Numerical integration methods are used to approximate the otherwise continuous differential equations governing the behaviour of each particle. The basic theory of the integration methods was covered in Section 2.5 in Chapter 2, while this particular chapter will focus on the implementation of these methods and discuss the advantages and disadvantages of using each. Five methods will be explored; the Euler, RK2, RK4, Leapfrog and Euler-Cromer algorithms. These methods are covered in Section 5.2.

Both these methods are used extensively in all or most of the particle system implementations, and work independently of the particle physics methods. As a result they are discussed separately from the particle system implementations, which are instead covered in Chapter 6.

## 5.1 Spatial Grid Algorithm

Collision detection between particles in a large particle system has always been a major problem in computer graphics. In the worst case scenario, every particle collides with every other particle within a single time step, making the computational complexity of the collision detection algorithm $O(n^2)$, where $n$ is the number of particles in the system. However, such a naive collision detection algorithm assumes all particles could potentially collide with any other particle regardless of their positions relative to each other in the simulation space.

*Spatial grid-based collision algorithms* [126] [127] can solve this problem, reducing the number of interactions to only those performed in the local space of the particle, not the entire

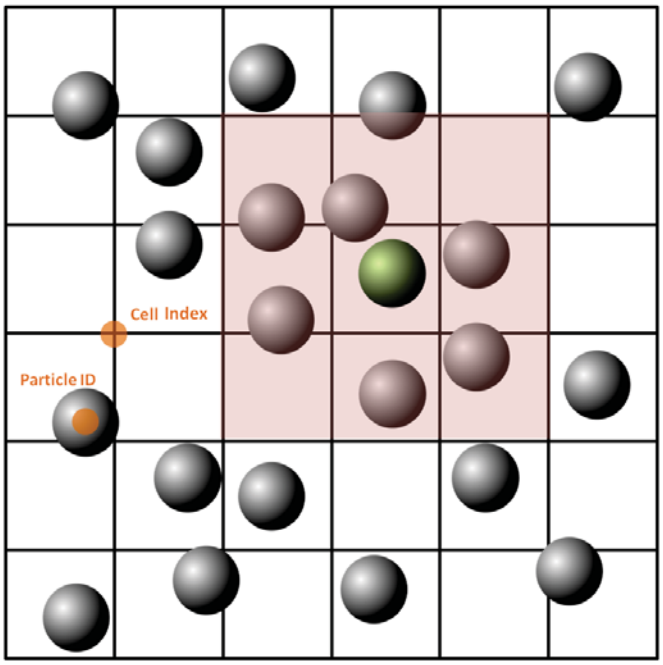


*Figure 5.1: Particle-to-particle interactions utilizing a spatial grid. Here the particles which are important are those contained within the red-coloured cells only. Particles are considered within a certain cell based on the center point position. These are the particles which will be tested for collisions with the current particle of interest (coloured green in this instance).*

simulation space. In recent years there have been many solutions [45] [31] [46] [41] proposed to this problem through the use of spatial grid approaches to reduce the time needed to compute collisions in large particle systems.

### 5.1.1 Spatial Grid-based Collision Detection

The simplest approach to grid-based collision algorithms [126] [127] is to apply a uniform grid to the simulation, allowing the *id* of each particle to be mapped to a specific cell in the grid. The particle's position is used to calculate a *cell index* value which determines which cell in the grid the particle is currently in. When calculating collisions between particles, only the particles within the same cell as the current particle, as well as those within neighbouring cells, are considered. All other particles are ignored. This achieves significant reduction in the number of particle-particle collisions needing to be calculated within any one timestep. See Figure 5.1 for an illustration of this method.

It should be noted that while a standard uniform grid is sufficient for speed-ups in most cases, it is by no means the only grid-based approach. Harada et. al. [45] proposed an approach using "sliced" two-dimensional planes to divide up the computational space, improving the memory efficiency of the simulation when compared to using a uniform grid by reducing the large number of empty grid cells that would inevitably occur when the particle distribution became irregular within the simulation space. Dobashi et. al. [31] introduce a method of

overlapping grids, solving problems arising when a uniform grid is not a sufficient enough resolution when dealing with interactions with complex rigid-body objects. These methods are more efficient when used in these specific simulation contexts, but for testing many different simulation models and the comparisons between them, using a uniform grid would provide more reliable comparable results during real-time simulations when it is not certain of the exact behaviour and subsequent distribution of the particles within the simulation space.

Grid-based collision detection approaches can further be improved by *sorting* all particles by their cell indices. Oliveira et al. [97] describe three sorting algorithms for use with grid-based particle collision detection methods, including a standard grid-based approach with no adaptive sorting. This approach was also used in Simon Green's particles demo [41]. Depending on the timestep used, an adaptive sorting algorithm might be preferable, as it would be unlikely that particles would move from one spatial grid cell to another in consecutive iterations if the timestep is relatively small. When sorting particle arrays, the sort is based on the cell index, so that particles with the same cell indices are contiguous in the array [97].

The spatial grid algorithm used in this thesis involves firstly selecting grid dimensions such that any object should not intersect more than eight cells in the grid. The size of the cell will vary depending on the simulation that is using the spatial grid. For a spring-mass model intending on utilizing a soft-body collision approach, the size of the cell should be no larger than twice the particle's radius, while if using a Smoothed Particle Hydrodynamics approach, the size of the cell should be the width of the smoothing kernel. Setting the dimensions to these sizes seeks to reduce the number of element interactions to the most efficient amount - ideally, keeping at most one particle per grid cell will produce the minimum number of interactions between particles in the local environment. Specifying the cell size is critical to the performance of the system; it is not guaranteed that the mapping of grid cells to particle ids will be unique [126], and every time multiple particles are mapped to the same cell, additional collisions will need to be computed. Ultimately this is unavoidable in numerically-integrated simulations, as the discrete timestep will allow some particles movement into cells which they otherwise would not be able to go, although this can be minimized by specifying a good cell size (as above) and a reasonably small timestep. In general, the smaller the number of particle ids per cell index, the greater the performance of the system.

Once the particle ids and cell indices have been mapped and sorted, they are scanned to determine the *first* and *last* particle ids which have the same cell indices. The result is two arrays of size $d_x \times d_y \times d_z$, one containing the ids of the first- and one containing the ids of last particles contained in each cell. Given a cell index, it is then possible to find the group of particles in the sorted particle array which are inside the particular cell. When processing collisions, the cell index is calculated using the particle's position, and additional neighbouring cell indices are also included. The particle ids which are mapped to these cell indices are the particles with which collisions are possible. The overall method of the grid-based collision algorithm just described is shown in Algorithm 5.

The general approach Algorithm 5 takes has been chosen when implementing particle interactions in parallel. Although non-parallel implementations could simply use dynamic arrays

**Algorithm 5** General method for spatial grid particle interactions

Given array P of *n* particles
Given array K of cell index / particle id pair, K.*cell_index* and K.*id*
Given $d_x$, $d_y$ and $d_z$ grid dimensions
//Calculate the cell index for each particle in the array
**for** *i* = 0; $i < n$; *i*++ **in parallel do**
  $pos_{x,y,z}$ = cell coordinates for P[$i$]
  *cell_index* = *cellfunc(*$pos_x, pos_y, pos_z$*)*
  K[$i$] = (*cell_index*, $i$)
**end for**
//Sort particles by cell index so that all particles with same cell index are together
**for all** $i$ in K **in parallel do**
  Sort K such that K[$i$].*cell_index* $\leq$ K[$i$ + 1].*cell_index*
**end for**
Given *first* and *last* arrays of size $d_x$ * $d_y$ * $d_z$
Reset *first* and *last* to default value D
//Find the particle ids of the first and last particle within each cell
//Use shared memory B of blocksize+1
**for** *i* = 0; $i < n$; i++ **in parallel do**
  *cell_index* = K[$i$].*cell_index*
  Neighbour index *n_index* = K[$i$-1].*cell_index*
  Synchronize threads
  **if** *cell_index* $\neq$ *n_index* OR $i == 0$ **then**
    *first*[*cell_index*] = $i$
    **if** $i > 0$ **then**
      *last*[*n_index*] = $i$
    **end if**
  **end if**
**end for**
//Interact only between particles in each cell and neighbouring cells
**for** *i* = 0; $i < n$; *i*++ **in parallel do**
  $pos_{x,y,z}$ = cell coordinates for P[$i$]
  **for all** neighbouring cells $pos_{x-1,y-1,z-1}$ TO $pos_{x+1,y+1,z+1}$ **do**
    *cell_index* = *cellfunc(*$pos_x, pos_y, pos_z$*)*
    Let *f* = *first*[*cell_index*]
    **if** $f \neq D$ **then**
      Let *l* = *last*[*cell_index*]
      **for** *j* = *f*; $j \leq l$; *j*++ **do**
        **if** $i \neq j$ **then**
          Interact particles P[$i$] and P[*j*]
        **end if**
      **end for**
    **end if**
  **end for**
**end for**

to store particle data by cell index and then process interactions between particles only in this list and neighbouring cell lists. This is difficult to do in parallel implementations because allocating memory dynamically on the device is very slow. Using Algorithm 5, there is no need for memory re-allocation, and sorting the particles reduces the search space when processing neighbour interactions. Additionally, it improves the spatial locality of the data, making cached memory access more efficient.

### 5.1.2 Sorting Considerations

When using small timesteps, particle positions may not change significantly between consecutive simulation steps, and it becomes inefficient to sort the entire particle array when the array is already *nearly sorted*. To improve this method, Oliveira et. al. [97] propose a adaptive sorting approach called a *split and merge* strategy which splits the particle arrays into two parts, one part containing known already-sorted particles and a second smaller part containing particles which need to be sorted. A particle will need to be re-sorted if its cell index has changed since the previous iteration.

A split-merge sorting strategy was tested against the standard THRUST `sort_by_key` implementation to see if there were any benefits in practice when using this sorting method. In theory, if the timestep is small enough, there would be so few particles which have moved outside their current cell into a new one that sorting the entire arrays of particle data would become inefficient compared to sorting the very small portion of particles which are not already sorted. The cell index arrays (K.*cell_index* as per Algorithm 5) is maintained between successive calls of the spatial grid update function, and is compared against the ids of the particles. If particle ids have cell indices which are different than the cell indices of the previous iteration, they must be sorted. Otherwise, they are ignored as they are already in sorted order. Performing an *exclusive prefix-sum* scan operation is used to find the individual cell index positions and total number of cell indices which need to be sorted, and then using a permutation operation to split the array into two manageable blocks, one already sorted and a much smaller one unsorted.

Oliveira et. al [97] suggest performing a second scan backwards to determine the ids of the already-sorted particles as well. A more efficient approach is described by Harris et. al [47] when using prefix-sums to develop an efficient radix sorting algorithm in CUDA, as performing entire scans of the array can be costly when the arrays are relatively large. This more efficient approach is to use the *inverse* of the boolean sorted data to permute only the *already-sorted* parts of the array. The efficiency is increased because only the already-sorted elements in the array need to preserve their order, while non-sorted elements need only be separated from already-sorted ones and their order not preserved, as they will be subsequently sorted anyway.

In practice, the split-merge sort approach only performed better than the standard approach for very small numbers of particles (up to $2^{13}$). This is due to the better-suited THRUST sorting algorithm - the THRUST `sort_by_key` function calls five CUDA kernels; two *upsweep* (reduction) kernels, two *downsweep* kernels and a single scan. Overall, these kernels execute faster than the prefix-sum algorithm, and time is only made up in small system sizes when it

```
__global__ void calc_cells(uint *particle_id, uint* cell_indices,
    float4 *pos, float3 cell_dim, uint numParticles){

    //Usual index calculation based on threads and blocks
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if(index >= numParticles){return;}

    float4 p = pos[index];

    //Find position in the spatial grid and calculate cell index
    int3 grid_pos = calcCellPos(make_float3(p));
    uint cell_index = calcCellIndex(grid_pos, cell_dim);

    //Output cell index and particle id
    particle_id[index] = index;
    cell_indices[index] = cell_index;
}
```

*Listing 5.1: Calculating the cell index and particle id pairs when initializing the spatial grid.*

becomes inefficient to scan such small-sized arrays on the GPU when compared to the CPU. Chapter 7 briefly touches on situations which may prefer the particle system size to be reduced, but in general the simulation of complex particle systems require far larger numbers of particles to produce a quality result. For this reason the Thrust sorting method is preferable to the split-merge strategy implementation.

### 5.1.3 Spatial Grid Implementation

Implementing spatial grid collision detection in CUDA is relatively straightforward as the method is easily parallelized and works well with the parallelized particle integration methods. The implementation of this spatial grid was based on the particles demo implementation in the CUDA Toolkit [41]. There are three main operations involved in parallelizing this method:

- Calculation of cell indices per-particle, used to determine which particles reside in the same cell and/or neighbouring cells in the spatial grid
- Sorting particle data such that neighbouring particles in world space are contiguous in the sorted array, allowing for easy calculation of the particle ids which represent the first and last particles in the sorted array which have the same cell index
- Performing collisions between particles within the current cell as well as neighbouring cells

Calculation of the cell indices per-particle is a simple function of the particle's current position, as explained in Algorithm 5. Listing 5.1 demonstrates this as a CUDA kernel.

The method of sorting was already analyzed in Section 5.1.2. The `first` and `last` arrays shown in Algorithm 5 are populated simply by looking at each element in the cell indices array and comparing it to the previous element. Since the cell indices are in sorted order, the current particle will be the first particle in the cell if the particle before it has a different cell index.

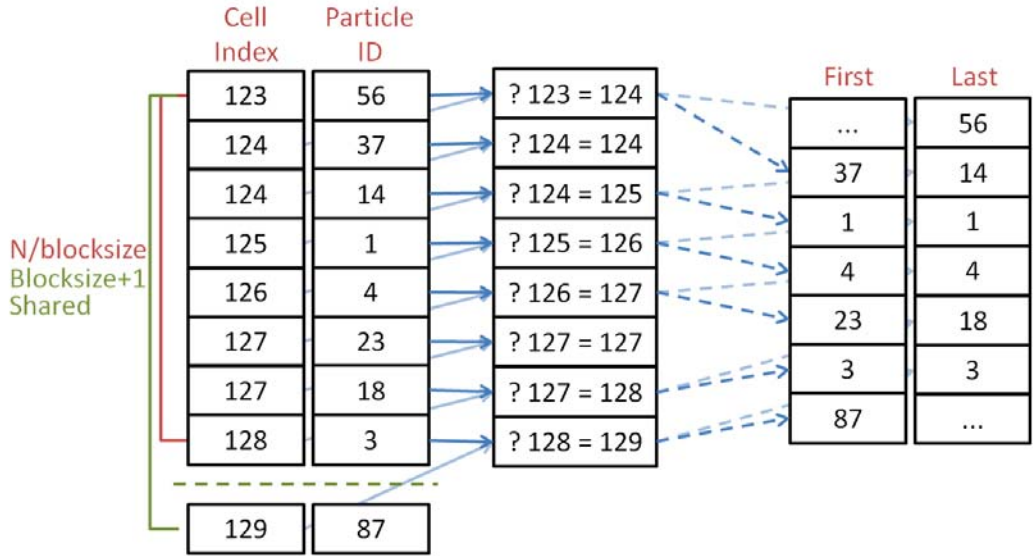


*Figure 5.2: Diagram showing how the first- and last- particle id arrays are populated using CUDA. All particle spatial grid data - cell indices and particle id arrays, which have been sorted based on the cell index, are divided into blocks and loaded into shared memory. For each block, every particle's cell index is checked against the previous one. If the cell index is different, the particle with the corresponding id must be of the first particle in this particular cell, and the previous particle in the sorted list must be the last particle in the cell with the previous cell index. Shared memory size of blocksize+1 elements allows an additional cell to be checked outside of the current block, to determine if the final particle in the block is also the last for the corresponding cell index.*

Similarly, the previous particle must then also be the last particle in the sorted array which has its particular cell index. In CUDA, this can be performed efficiently by loading the cell index array into shared memory, storing shared cell indices one element to the right and making sure to include the right-most particle from the previous block. This process is shown in Figure 5.2.

When calculating collisions, the position of the cell each particle resides is calculated and then a list of the other 26 neighbouring cells is generated. These neighbours should include not only the six nearest-neighbour cells in both positive and negative directions on each axis ($x$, $y$ and $z$), but also the cells neighbouring diagonally, as it is possible that particles classified with these cell indices could also partially lie within the current cell. Once this list has been calculated, collisions are performed between the current particle and every particle classified as having the same cell index, and this is repeated for each neighbouring cell. The particles with which the current particle needs to collide are found using the `first` and `last` arrays calculated in the previous step. Implementations of the spatial grid vary depending on the types of forces which are to be calculated. Therefore, collision implementations for the different particle systems will be discussed in Chapter 6.

Once all the forces have been calculated, the spatial grid algorithm is complete and the simulation moves on to the integration method where the resulting force is used to further the particle velocity and position. This will be explained in the next section.

## 5.2 Numerical Integration

In Section 2.5, numerical integration methods were introduced. They are used to computationally solve the continuous differential equations governing the behaviour of particles. When calculating the positions and trajectories of particles, it is important to consider higher order integration methods when dealing with large particle systems, especially systems which use inverse-square laws [52], such as Coulomb's law for electrostatic force. Numerical integration methods essentially approximate otherwise continuous equations with discrete solutions - this introduces some degree of error, and using higher order integration methods ensure that the simulation model is a reasonably accurate representation. This means minimizing the error introduced by these methods, however care must also be taken when rendering simulations in real-time. High-order integration methods generally result in better numerical precision, but at the cost of requiring more integration function calculations [52] resulting in a considerable increase in computational costs over lower-order integration methods. When applying this problem to real-time rendering, it brings up the question of whether ultra-small differences in error will even be noticeable visually when the system is rendered.

There are five different integration methods considered in this thesis for integrating particle systems in real-time, the Euler method, Second order Runge-Kutta (RK2), 4th order Runge-Kutta (RK4), Leapfrog method and the Euler-Cromer method. Comparisons will be made between the methods in relation to real-time simulation. In order to perform the higher order integration methods, all the above steps (spatial grid, first/last particles within cell, collisions) need to be executed multiple times to obtain the correct force for each stage in the integration. In the case of the RK4 method, the steps must be executed four times each before the new particle positions and velocities may be outputted.

As with all the previous steps, the parallelization of the integration methods is done per-particle and there is no need for communication between different particles, as the acceleration or force calculations have already been performed. It is most beneficial for the integration method to be completely separate from the other simulation kernels, so that different integration methods can be substituted in with no effect on the rest of the simulation.

### 5.2.1 Integration Implementation

Implementation of the Euler method is straightforward, only requiring a single integration of the position and velocity differential equations to obtain the solution. The implementation of this as a CUDA kernel is shown in Listing 5.2. It should be noted that all positions, velocities and accelerations are stored as `float4` variables, although they are technically only 3-dimensional vectors. `float4` is used instead of `float3` so that it the global memory accesses do not break alignment. For optimal coalescing, the width of the arrays stored in global memory (as well as the width of the thread block) should be a multiple of the warp size [91], so that when warps access words in memory, the number of memory transactions that include memory not accessed by the threads in the warp is minimized. Since the warp size is 32 threads,

```
__global__ void euler_integration(float4* in_out_pos, float4* in_out_vel,
    float4* in_acc, uint numParticles, uint dt){

    //Standard index calculation
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    //Get the current particle data
    float3 pos = make_float3(in_out_pos[index]);
    float3 vel = make_float3(in_out_vel[index]);
    float3 acc = make_float3(in_acc[index]);

    //Standard Euler Method
    //Vn+1 = Vn + (a)dt
    //Pn+1 = Pn + (Vn)dt
    pos += vel * dt;
    vel += acc * dt;

    //Output position and velocity
    in_out_pos[index] = make_float4(pos, 1.0f);
    in_out_vel[index] = make_float4(vel, 1.0f);
}
```

*Listing 5.2: Euler Integration*

a `float4` data structure is preferable as it is already a power-of-two size. If coalescing is optimized like this, there are better performance results from the simulation.

The Euler-Cromer method varies slightly from the standard Euler in that it calculates the velocity at the end of the timestep first, and then using this velocity to calculate the endpoint position. Using the Euler-Cromer over the standard Euler results in a much more stable integration method at virtually no additional computational cost. The implementation is shown in Listing 5.3.

Implementing RK2 is very similar, however it requires that the acceleration is calculated twice, once using a timestep of $t/2$, and then again for the final position and velocity calculations using the midpoint calculation of the force as the approximation over the entire time interval. The spatial grid, sorting and collision portions of the simulation must be completed twice because, for example, a particle might have moved into a different spatial grid cell when using timestep $t$ than it would when using timestep $t/2$.

Integration using RK2 is performed in two different steps. As shown in Listing 5.4, the original position and velocity arrays are stored separately in the $in\_out$ arrays and kept for use in the second step. The position and velocity is then calculated at the midpoint $t/2$, and stored in secondary $out\_mid$ arrays. The velocity at the mid point is needed for the second step, while the position at the midpoint is needed to calculate the midpoint acceleration, using the necessary force calculations. For example, the repelling force in a spring-mass system depends on the position of the two particles colliding - to find the acceleration at the midpoint then, the two particles must first be moved to the midpoint (calculate the midpoint position). Once the midpoint acceleration is calculated, it is passed as the input $in\_acc\_mid$ in the second RK step. Having this midpoint acceleration allows the final output position and velocity to be calculated

```
__global__ void euler_cromer_integration(float4* in_out_pos,
    float4* in_out_vel, float4* in_acc, uint numParticles, float dt){

    //Standard index calculation
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    //Get the current particle data
    float3 pos = make_float3(in_out_pos[index]);
    float3 vel = make_float3(in_out_vel[index]);
    float3 acc = make_float3(in_acc[index]);

    //Euler-Cromer integration
    //Pn+1 = Pn + (Vn+1)dt
    //Vn+1 = Vn + (A)dt
    vel += acc * dt;
    pos += vel * dt;

    //Output position and velocity
    in_out_pos[index] = make_float4(pos, 1.0f);
    in_out_vel[index] = make_float4(vel, 1.0f);
}
```

*Listing 5.3: Euler-Cromer Integration*

using the midpoint formula. This second step is shown in Listing 5.5.

Using the RK2 method over the Euler method has the upside of getting results which are more accurate and stable, as will be shown later. However, the method requires a second pass through the particle force data calculation functions as well as the spatial grid kernels. Additionally, RK2 requires extra memory to store the midpoint values. Section 5.2.2 will discuss this further.

The RK4 implementation requires four steps - four different RK values of the velocity and force are required, so firstly extra memory must be allocated to account for this. The first three RK4 steps are similar to the RK2 implementation, calculating the various rates of change depending on the timestep (refer to Equation 2.11 for these rates of change) and storing these for use in the final step. In the first step, the original position, velocity and acceleration is stored, while in the second and third steps only the velocity and acceleration are required. The fourth step is performed once all four points on each function have been calculated, and the weighted average of the rates is found to determine the final position and velocity of the particle. This is shown in Listing 5.6.

The leapfrog integration kernel is slightly simpler than the RK variants as, although it is a second-order method, it only requires a single evaluation per simulation step, similar to first-order methods such as the Euler and Euler-Cromer. It requires extra memory to store the velocity values from the previous timestep. Listing 5.7 shows an implementation of the method.

Note that only the original velocity half a timestep behind the original position ($vel\_half$) is used to calculate the next position, and is then "leapfrogged" over the original position as $vel\_new$. Meanwhile, $vel\_current$ is evaluated as the average of the two velocities for an

```
__global__ void rk2_step1(float4* in_out_pos, float4* in_out_vel,
    float4* out_pos_mid, float4* out_vel_mid, float4* in_acc,
    uint numParticles, uint dt){

    //Standard index calculation
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    //Get the current particle data
    float3 pos = make_float3(in_out_pos[index]);
    float3 vel = make_float3(in_out_vel[index]);
    float3 acc = make_float3(in_acc[index]);

    //Midpoint step
    //Pn+1/2 = Pn + (Vn)dt/2
    //Vn+1/2 = Vn + (a)dt/2
    float3 pos_mid = pos + vel * dt/2;
    float3 vel_mid = vel + acc * dt/2;

    //Output mid position and velocity
    out_pos_mid[index] = make_float4(pos_mid, 1.0f);
    out_vel_mid[index] = make_float4(vel_mid, 1.0f);
}
```

*Listing 5.4: RK2 Integration, first step*

```
__global__ void rk2_step2(float4* in_out_pos, float4* in_out_vel,
    float4* in_vel_mid, float4* in_acc_mid,
    uint numParticles, float dt){

    //Standard index calculation
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    //Midpoint values
    float3 vel_mid = make_float3(in_vel_mid[index]);
    float3 acc_mid = make_float3(in_acc_mid[index]);

    //Original values
    float3 pos = make_float3(in_out_pos[index]);
    float3 vel = make_float3(in_out_vel[index]);

    //RK2 Integration using midpoint
    //Pn+1 = Pn + V(mid) dt
    //Vn+1 = Vn + A(mid) dt
    pos = pos + vel_mid * dt;
    vel = vel + acc_mid * dt;

    //Output position and velocity
    in_out_pos[index] = make_float4(pos, 1.0f);
    in_out_vel[index] = make_float4(vel, 1.0f);
}
```

*Listing 5.5: RK2 Integration, second step. The acceleration at the midpoint is calculated from the midpoint position between step1 and step2.*

```
__global__ void rk4_step4(...load in all RK4 arrays...,
                          uint numParticles, uint dt){

    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    //Get the RK values
    float3 pos = make_float3(in_pos[index]); //Original pos
    float3 vel = make_float3(in_vel[index]); //Original vel
    float3 k1_a = make_float3(k1_acc[index]);
    float3 k2_v = make_float3(k2_vel[index]);
    float3 k2_a = make_float3(k2_acc[index]);
    float3 k3_v = make_float3(k3_vel[index]);
    float3 k3_a = make_float3(k3_acc[index]);
    float3 k4_v = make_float3(k4_vel[index]);
    float3 k4_a = make_float3(k4_acc[index]);

    //RK4 - Yi+1 = Yi + 1/6(k1 + 2k2 + 2k3 + k4) * dt
    pos += ( (vel + 2*k2_v + 2*k3_v + k4_v) / 6.0f) * dt;
    vel += ( (k1_a + 2*k2_a + 2*k3_a + k4_a) / 6.0f) * dt;

    //Output final position and velocity
    out_pos[index] = make_float4(pos, 1.0f);
    out_vel[index] = make_float4(vel, 1.0f);
}
```

*Listing 5.6: RK4 Integration kernel (final step)*

approximation of the velocity at time $t + 1$ rather than $t + 1/2$ or $t + 3/2$. This is the velocity value used to calculate the particle acceleration in the next iteration. The leapfrog method works particularly well with this particle system, as it only requires a single pass through the particle force calculation functions because only a single acceleration value is required for each particle. Additionally, because it is a second order method, it is more accurate than first order methods such as the Euler variants. This is discussed in more detail in the next section.

### 5.2.2 Integration Method Comparison

The suitability of the different integration methods can be compared by calculating the energy of the particle system every iteration of the simulation and comparing it to the energy of the previous iteration. Since energy should be conserved, any change in the total energy of the system will be due to the error introduced by the numerical approximation. For particle systems in general, the total energy of each particle in the system can be given by the formula in Equation 5.1.

$$E_{total} = E_{kinetic} + E_{potential} \tag{5.1}$$

$$E_k = \frac{1}{2} m |\vec{v}|^2 \tag{5.2}$$

```
__global__ void leapfrog_integration(float4* in_out_pos, float4* out_vel,
    float4* out_vel_half, float4* in_vel_half, float4* in_acc,
    uint numParticles, float dt){

    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= numParticles) return;

    float3 pos = make_float3(in_out_pos[index]);            //Pn
    float3 vel_half = make_float3(in_vel_half[index]);      //Vn-1/2dt
    float3 acc = make_float3(in_acc[index]);                //A dt

    //Integration using Leapfrog method
    //Vn+1/2 = Vn-1/2 + (A)dt
    //Vn+1 = ( (Vn+1/2) + (Vn-1/2) ) / 2
    float3 vel_new = vel_half + acc * dt;
    float3 vel_current = (vel_half + vel_new) * 0.5f; //V(dt) approx

    //Update position of particle
    pos += vel_current * dt;

    //Output final position and velocities
    in_out_pos[index] = make_float4(pos, 1.0f);
    out_vel[index] = make_float4(vel_current, 1.0f);
    out_vel_half[index] = make_float4(vel_new, 1.0f); //Next Vn-1/2dt
}
```

*Listing 5.7: Leapfrog integration. Notice the current value for the half-timestep velocity $vel\_half$ is stored in the output array as $out\_vel\_half$ to be used in the next iteration, while $vel\_current$ represents the average of both half-step velocity values as the current velocity and is stored in the output array $out\_vel$. This is used for later force calculations which require velocity as an input, such as viscous force calculations.*

The kinetic energy of a particle is given by the formula shown in Equation 5.2. The potential energy of a particle however is slightly different as this is dependent on the type of particle system being simulated. Newtonian gravitational potential energy of the interaction between a pair of particles $i$ and $j$ is can be calculated using the formula in Equation 5.3 [52].

$$U = \frac{-Gm_i m_j}{r} \tag{5.3}$$

where $G$ is the gravitational constant, $m_i$ and $m_j$ are the masses of the two particles and $r$ is the scalar distance between them. This equation is similar to the equation for the potential energy in an electrostatic field, where the energy is determined by the charges of the two particles $q_i$ and $q_j$, the difstance $r$ between them, and the constant $k_e$, or Coulomb's constant. This is the energy equation used for calculating the potential energy in the plasma system. It is shown in Equation 5.4.

$$U = k_e \frac{q_i q_j}{r} \tag{5.4}$$

When modelling a system where the gravitational potential energy is calculated relative to the Earth's surface (i.e. the potential energy of an elevated object), such as the water simulation described later in 7, the formula is given by Equation 5.5, where $m$ is the mass of the particle, $g$ the magnitude of the gravitational force and $h$ the height of the particle.

$$U = mgh \tag{5.5}$$

In a spring-mass system, potential energy also exists in the compressing and extending of the springs. The potential energy of a compressed spring is given by the formula shown in Equation 5.6. Here the energy is based on the spring constant $k_s$ and the distance the spring has been compressed $x$ from the rest position. The further the spring has been compressed, the greater the potential energy.

$$U = \frac{1}{2}k_s x^2 \tag{5.6}$$

If an integration method introduces error into the system where the position has moved a lot further than otherwise would be expected, the change in potential energy will be greater than expected (for example, a spring will be compressed a much greater distance). Similarly, error introduced causing the particles to move faster or slower than otherwise expected will cause changes in the overall kinetic energy of the system. By adding up the kinetic and potential energy of all particles, the total energy of the system can be monitored over time and observed for changes introduced due to error.

Good energy conservation levels will be more difficult to achieve depending on the type of simulation. If the particles in the system are constantly moving around at relatively fast velocities, the error introduced will be relatively large compared to a system which does not move as much. An example of such a system might be a "wave" type system where water particles are constantly crashing against boundary walls and rising and falling from greater heights. Such systems are referred to as *active* systems. Systems which are almost at an equilibrium state (for example, still water) will conserve energy a lot better but will still experience differences in energy conservation as the particles will still always be moving, only at a much smaller amount. These types of systems are referred to as *calm* systems.

The energy conservation of a standard spring-mass particle system for simulating water is shown as a graph in Figure 5.3. The energy is sampled 100 times over the course of 1000 simulation iterations, and it has been allowed to settle somewhat ( 2000 iterations for active system), as there are large amounts of energy introduced at the very beginning of the simulation due to initial randomized particle placings - as particles are randomly placed, two particles which are relatively close may immediate exert a strong spring force on each other.

For a calm system, the graph is shown in Figure 5.4. It should be noted that as the system is a calm system, the scale on the left hand side is significantly smaller than that of Figure 5.3,

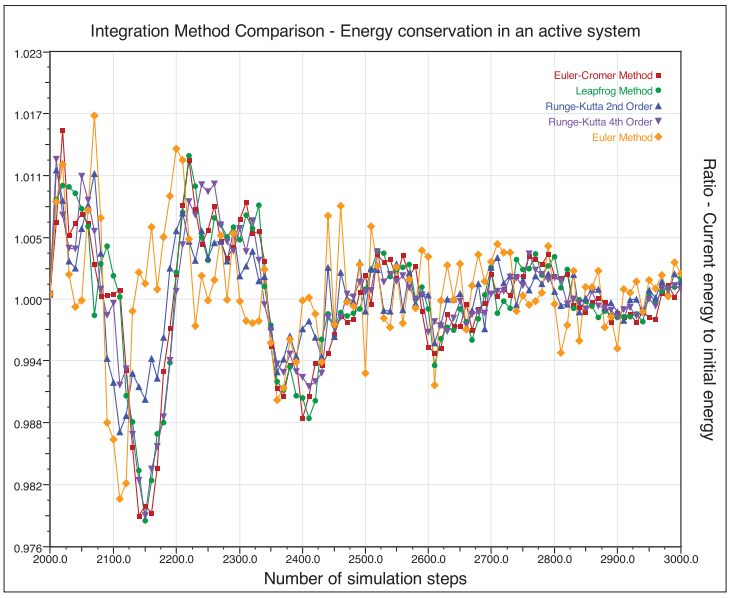


*Figure 5.3: Comparing the energy conservation of an active water particle system using different integration methods. All integration methods manage to remain relatively stable, but there are clearly varying degrees of error for each method. The Euler method contains the most error, followed by the RK2 method, although this has improved substantially. RK4, Euler-Cromer and Leapfrog methods conserve much more similar amounts of energy, although RK4 seems to have slightly more variation.*

due to the fluctuation in the energy levels being much lower.The reduced scale allows a better representation of the differences between the integration methods.

In both these cases the particle systems have been able to reach a relatively stable state, however this is not always the case. Figure 5.5 shows what happens when the integration method causes the particle system to become unstable. In this case, the particles collect so much energy and movement that they break out of the boundaries of the system. This simulation in particular differs from Figures 5.3 and 5.4 in that the timestep has been modified to a point that it causes instability in the Euler method - the others used a timestep which allows a stable system for *all* integration methods. Halfway through the simulation the Euler method causes the system to become unstable. Note that the error bars have been removed from this graph to make it more readable - error becomes very large as the instability in the Euler method increases.

In both the cases shown in Figure 5.3 and Figure 5.4 the energy has been measured over a single execution to demonstrate the possible fluctuation in energy of the system at any one time. Over multiple executions, the energy conserved is averaged over all iterations for each integration method, resulting in a wave-like shape to the graph. This is caused by the initialization of the fluid particle system. The particles are dropped from a height to the floor, where they consequently hit the ground and spread, hitting the walls and colliding with each other in a periodic manner. This causes additional error to be added to the energy of the system by the integration method, due to the larger than usual compressed spring potential energy as particles

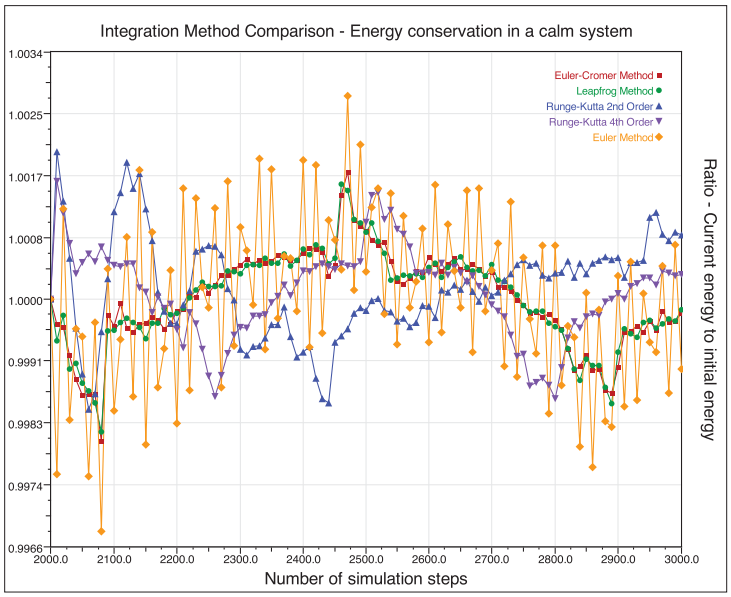


*Figure 5.4: Comparing the energy conservation of a calm water particle system using different integration methods. The Euler method is more clearly shown as an impractical method in this graph, as shown in the constant variation of the energy values. Leapfrog and Euler-Cromer methods maintain a fairly stable energy conservation. RK2 shows slightly more variation in the error than RK4, and RK4 seems to more closely follow the energy conservation levels of the Leapfrog and Euler-Cromer, although both RK methods are significantly better than the standard Euler method.*

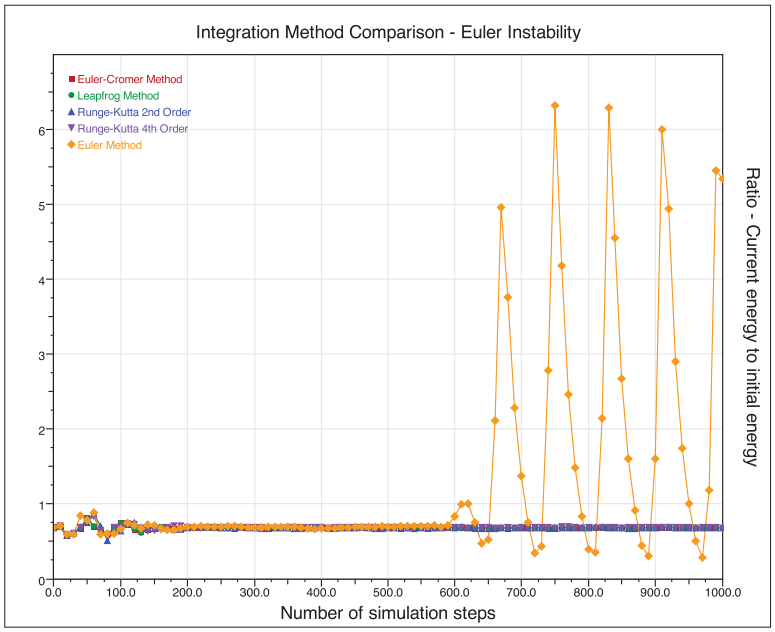


*Figure 5.5: Comparing the energy conservation of a water particle system. Using the Euler integration method in this situation caused the particle system to become unstable. This particular energy data was gathered from the very beginning of the simulation to demonstrate the exact time frame. Notice there is some initial energy fluctuation due to the initialization parameters but this settles down.*

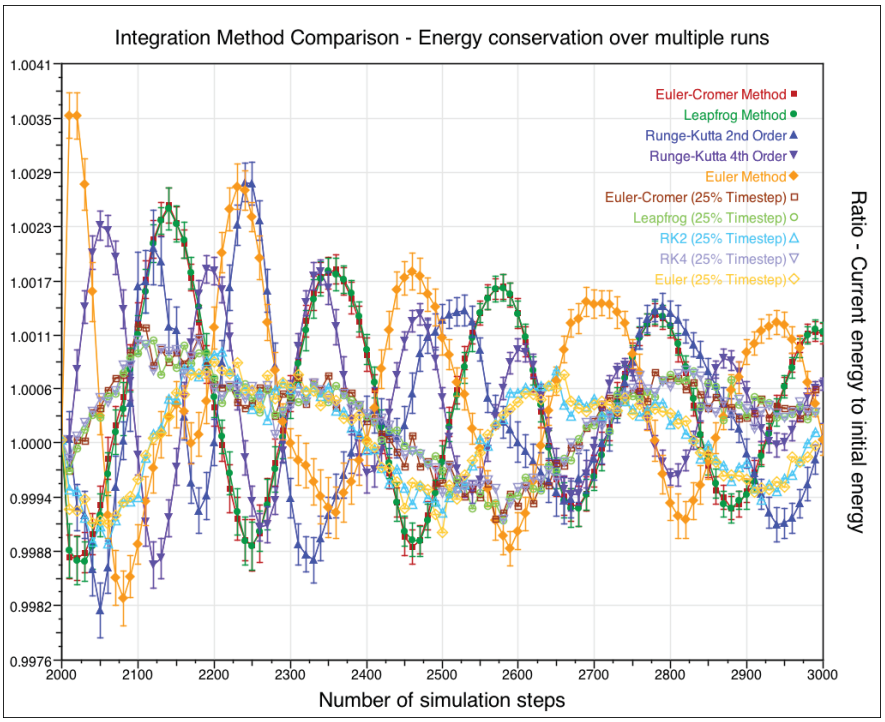


*Figure 5.6: Conservation of energy for different integration methods averaged over 50 runs. The simulation has been run again at a decreased size timestep to show how the amount of energy added by the integration method (error) decreases. A 25% timestep is used, although because of the timestep reduction, the energy is sampled once over 4 timesteps to maintain the same execution time overall.*

periodically collide with each other and the environment. Despite this, the energy conservation remains very close to 1.0 for all methods, as a suitable timestep has been chosen.

Figure 5.6 shows this, also demonstrating the effect using a smaller timestep. Things to note in this graph is that the RK4 method in particular introduces the least amount of error. Additionally, the shape of the Leapfrog and Euler-Cromer methods are very smooth and cyclical, evident of the symplectic nature of these methods (i.e. they remain the most stable over time).

In addition to the energy conservation of the particles, the execution time of the different methods also needs to be considered. Although some methods might produce less error, if they are slower then they might still not be the optimum choice. Generally, higher-order integration methods require more computation steps than lower order integration methods.

The average execution times of a full simulation step using different integration methods with increasing numbers of particles in the system. The timing data is sampled over 50 runs and averaged over 1000 iterations similar to the energy conservation tests, where the system has been allowed to settle into a relatively stable state before timing begins. All time performance graphs are drawn using a log2 scale so that they are easier to read. The size of the particle system ranges from 16384 ($2^{14}$) particles in the smallest case, to 524288 ($2^{19}$) in the largest case. Figure 5.7 shows the execution times of both a calm and an active particle system. One thing to note here is that although the Leapfrog method is generally regarded as a second-order

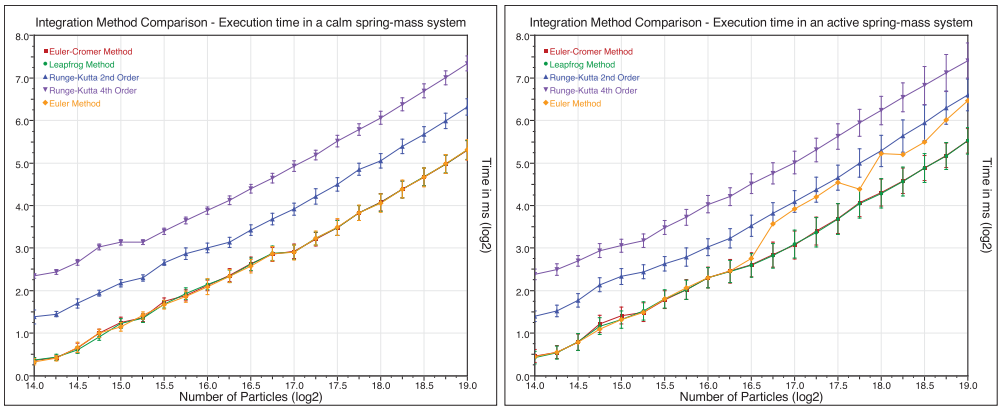


*Figure 5.7: Comparison of the execution times for a spring-mass model modelling a water particle system. The RK4 method is the most computationally expensive, followed by the RK2 method, while the other three first-order methods maintain a very similar execution time.*

method, it maintains the execution time of a first-order method. This comes only at the cost of additional memory. This makes the Leapfrog integration method particularly attractive to use for particle systems. Another thing to note is that in the situation of the active particle system, utilization of the Euler method results in instability of the system for larger numbers of particles, causing the execution time to fluctuate greatly and increase on average. Due to this high fluctuation, the variance in the Euler plot is particularly large and thus the error bars have been omitted as it overshadows the rest of the graph.

For a Smoothed Particle Hydrodynamics system, the results are similar, as shown is Figure 5.8. One thing to note here is that, similar to the spring-mass system, the Euler method again fluctuates greater than the other first-order methods in the case of the calm system, although in this case the timestep is small enough for it to stay relatively stable. The main difference between the spring-mass and SPH methods is that execution times vary across different system sizes when compared to the spring-mass model. This is discussed in greater detail in Chapter 6. As the particle system size increases, it becomes more important than ever to choose an efficient integration method.

A visual representation of the difference between the integration methods can be seen in Figure 5.9 as a number of different screenshots. The screenshots are taken 2000 iterations after initialization, where particles are dropped in a cluster in the middle of the simulation space, causing a wave-like behaviour to propagate from the center outwards to the edges when the fluid hits the floor. The particle system is an SPH system with $2^{17}$ particles and uses the same timestep. As shown in the figure, the Leapfrog and RK4 methods provide the smoothest flow of particles, while the RK2 method and especially the Euler method causes the particles to escape the fluid much easier due to the larger error introduced. This is not desirable behaviour particularly for a fluid simulation where particles should be inclined to stay together. Escape velocities need to be especially strong to cause this behaviour in a SPH particle system, as there are viscosity forces present which actively work to keep fluid particles together.

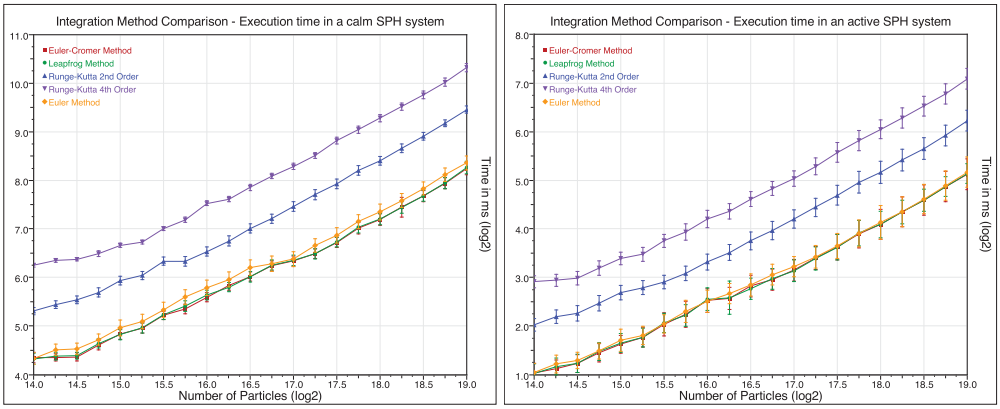


*Figure 5.8: Comparison of execution times for a SPH particle system of varying sizes. Unlike a spring-mass system, SPH calculates pressure values around areas in the fluid. Therefore the fluid distribution in a calm system is likely to have a higher number of particles concentrated in one place (the bottom of the container) and thus more pressure calculations must be performed. This causes the average execution time of an calm SPH system to be relatively higher than an active one. It should be noted that other factors, such as the smoothing kernel width, will also affect the computational complexity of this method. For this test these parameters have been kept constant.*

Decreasing the size of the timestep will decrease the error caused by all integration methods and in particular allow poor energy conserving methods to maintain a more stable simulation. However, this will require more simulation iterations to achieve arrive at the same state. This is this is an issue when simulating in real-time. While a single frame may be rendered over several timestep iterations, these iterations can be computationally expensive especially for large particle systems. Ideally, the integration method should allow for a timestep which provides good behavioural qualities of the particle system while minimizing the number of iterations per frame.

For example, using the RK4 method requires fewer iterations per frame as the behaviour of the system is fairly accurate. In contrast, the Euler method requires a must smaller timestep (and therefore more iterations per frame) to achieve the same behaviour at the same frame rate.

Taking all these factors into account, the Leapfrog algorithm is the best method to use for the particle systems featured in this thesis, for a number of reasons.

- It requires only a single pass of the particle physics kernels, so it is computationally inexpensive.
- It provides good energy conservation, allowing for more stable system behaviour over a long period of time.
- Is a second-order method, and therefore preferable over similar-performing first-order methods such as the Euler-Cromer.
- These factors minimize the number of iterations per frame and thus allow the system to efficiently perform in real-time.

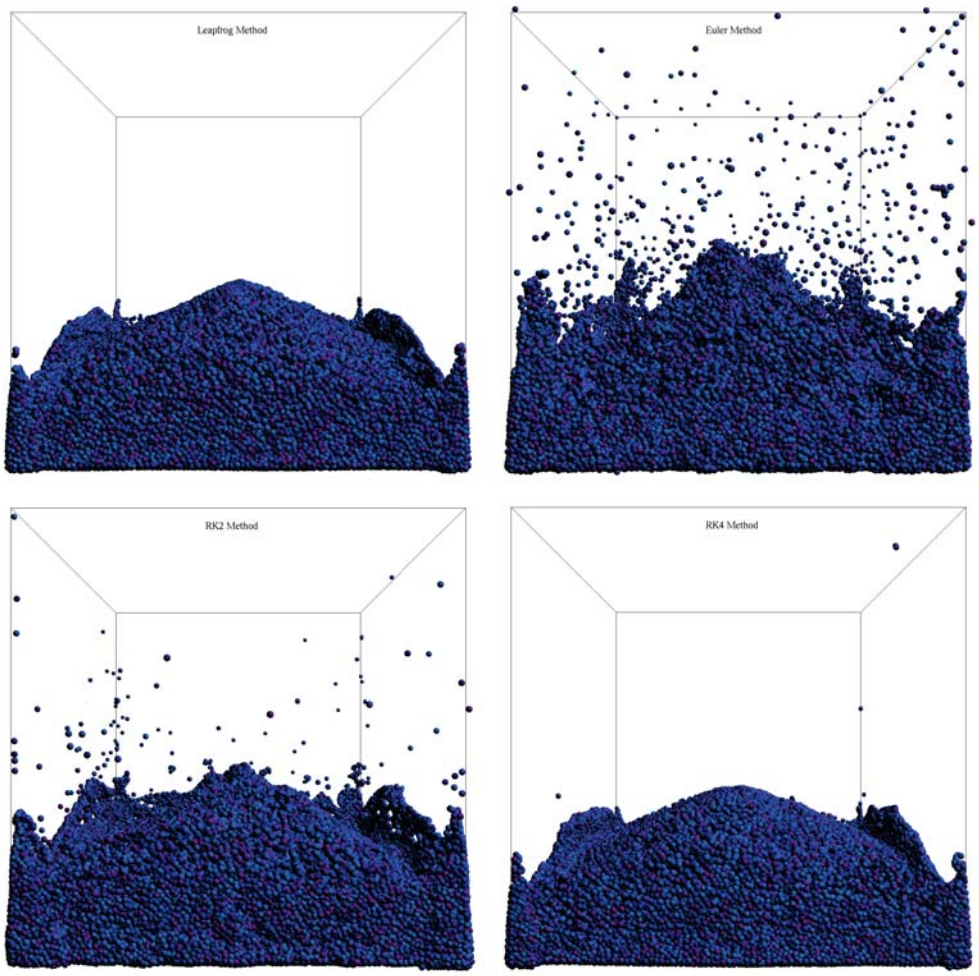


*Figure 5.9: Screenshot of the water particle system using different integration methods. It is rendered using point-sprite spheres. See Chapter 7 for more information on rendering methods. Top left: Leapfrog method. Top right: Euler method. Bottom left: RK2 method, Bottom right: RK4 method. The simulation involves initializing the particles randomly within a cube in the middle of the simulation space and dropping them to the ground.* $2^{17}$ *particles are used in this simulation.*

## 5.3 Summary

This chapter focuses on the implementation of the spatial grid and integration methods relevant to the particle system implementations used in this thesis. The integration methods are used in all systems, while the spatial grid method is used for all systems except for the non-interacting particle system. Symplectic integration methods such as the Leapfrog and Euler-Cromer methods are preferable as they conserve the most energy over a long period of time, whereas the higher-order RK4 conserves more energy between subsequent iterations. Additionally, higher-order integration methods require more computation resulting in slower execution times overall, and this can be detrimental when used in a real-time simulation. The next chapter will cover the core implementation of each of the individual particle systems in CUDA

and OpenGL - this includes the non-interacting system and the velocity-vortex method for fire, and the spring-mass and SPH systems for water simulation. Additionally there will be an additional "plasma" implementation, primarily for demonstrating the challenges involved with interacting with the spatial grid. The implementations of the spatial grid and the choices in the integration method are all central to the idea of simulation in real-time. The next challenge is making sure that the models themselves are also able to provide a realistic visual representation while also maintaining this interactive speed.

# CHAPTER 6

# IMPLEMENTATION

In Chapter 5, the integration methods and the spatial grid implementation was discussed. These methods are supplemental to the three different simulations - fire, plasma and water - and do not change much between them. The overall behaviour of each phenomena however, as well as the methods in which they are rendered, vary significantly, and each simulation involves different challenges in both behaviour physics and rendering methods. Care must be taken to account for these challenges especially when simulating in real-time.

Two methods used for fire simulation will be described. The first is a highly simplified method using a non-interacting particle system. The second uses a velocity-vortex particle model specifically designed for fire simulations. A non-interacting particle system is easy to implement and runs fast, but the simulation is not particularly interactive and the visual results are less than adequate for certain desired effects. In contrast, the velocity-vortex particle model allows the system to be highly interactive and produce much more realistic results, at the cost of being more computationally intensive.

The plasma system particle model uses electrostatic charge forces to pull or push particles together or apart. Forces are exerted over longer distances than usual and this results in issues when working with the spatial grid. Unlike the fire simulation models, the plasma system allows particles to interact with each other, resulting in further complex behaviour and intensity.

The water system uses two distinct particle system approaches. The first is a spring-mass model, where the particles apply soft-body collision forces on each other when the spring is compressed, and ignore forces due to spring extension as the particles move away from each other. The second uses a smoothed particle hydrodynamics (SPH) system to more realistically represent the overall behaviour of the liquid. The SPH system provides a much more realistic simulation at the cost of a higher computational cost than the spring-mass system.

The fire implementations will be explained in Sections 6.2 and 6.4 for the non-interacting and velocity-vortex methods respectively. In between these, Section 6.3 will describe the plasma implementation in relation to the interaction between particles in the spatial grid, as this is important to consider before moving on to the velocity-vortex simulation. Section 6.5

will describe the water simulation implementations including spring-mass and SPH methods, and provide some initial comparisons between the two methods.

## 6.1 Fire Introduction

Fire can be defined as a rapid, self-sustaining oxidization process of combustible gases ejected from a fuel source [6]. It is a challenging natural phenomena to simulate with computer graphics in real-time and in a realistic fashion, and can be used in a wide variety of applications including special effects in movies, video games, augmented reality and scientific visualization. It is a challenging phenomena to simulate because of its high complexity and turbulent nature. Highly realistic fire and flame simulations can be rendered using non-real-time processing methods, but this becomes much more difficult when trying to maintain the same high degree of realism while also rendering the simulation in real-time.

The simulation of fire and flame behaviour is a popular area of research in multiple disciplines [6] [134] [133] [56], and as such has produced a vast array of rendering techniques and behavioural methods to simulate and render the fire. Balci and Faroosh [6] use a spring-mass model to model flame kinematics, allowing external forces such as gravity and wind to be incorporated for added realism. The stated inspiration for the model being an old magician's trick called a "silk torch" for producing a fire-like illusion. Wei and Huang use a CUDA-based method similar to the one used in this chapter to render fire simulations in parallel using *Pixel Buffer Objects* (PBOs) in OpenGL [133], while Wei et al. also use a rendering style of texture splats in OpenGL (also very similar to the rendering style used in this chapter), while using the *Lattice Boltzmann Model* (LBM) to simulate the fire physics [134].

Horvath and Geiger use GPU-based rendering and refinement techniques to produce very high resolution fire simulations [56]. Lee et al. represent fire as an *evolving flame front* combined with a particle system approach for simulating a fires movement over a polygonal surface [69]. Melek and Keyser use fluid dynamics to model the motion of the fire while also simulating the combustion process of the burning fuel objects [82]. Other methods include rendering the fire include generating points on the surface of a *polygonal mesh* and using individual flame primitives to render the fire [10] as well as a graph-based approach by Zhang et al. using CUDA to solve the Navier-Stokes equations for fluid motion [145].

## 6.2 Non-interacting Fire Implementation

The first fire system particle model uses the simple non-interacting particle model featured in Chapter 4 Section 4.1. A non-interacting particle model means that particles do not interact with each other - therefore, much of the complexity needed to handle interactions between particles is not required and as such the system is very fast. All that is required is efficient parallelization of the particle behaviour code.

Each fire particle has the standard position, velocity and acceleration values stored in a floating point array. Because this is a non-interacting particle model, no spatial grid imple-

mentation is required. However, in order to simulate somewhat realistic behaviour, particle creation and destruction must be enabled. The current *life* and *decay rate* of each particle are stored along with the usual attributes. This life and decay rate is randomly initialized at the beginning of the simulation, with an underlying bias depending on the particle's initial position - this allows the shape of the particle system to change by using values which will create the desired behaviour. For example, if the particles are initialized in a sphere, then when a particle is initialized closer to the center of a "flat" circle (where the circle extends in the $x$ and $z$ directions) within the sphere, the maximum life of a particle is increased as the decay rate decreases - depending on how close to the center of the circle each particle is created. This gives the desired effect of particles nearer the middle of the flame to burn longer, resulting in a more "flame-like" shape.

Overall, the method of the non-interacting particle system follows two main steps:

- Creation and destruction of particles
- Advancing position and velocity (integration)

The integration methods have already been described in Chapter 5. The creation and destruction part serves as the main portion of the behavioural governing code for the simple non-interacting fire system, and can be summed up in a single CUDA kernel, as shown in Listing 6.1.

The basic functionality of the non-interacting particle system kernel is simply to decrement the life of the particle with the current timestep based on the particle's decay rate A slower rate of decay means the particle will live longer. As particles nearer the center of the flame have lower decay rates, this should make these particles live longer and thus rise higher, creating the flame shape. Position, velocity and acceleration are stored as `float4` variables for coalesced memory access efficiency, however $w$, the fourth homogeneous coordinate of the position vector, is not used. So rather than creating entire new float arrays for the particle life and decay data, they are simply stored as part of the position and velocity arrays (life and decay respectively). Every iteration of the simulation, each particle kernel checks if the particle life has reached zero - if it has, this means that the particle must be destroyed and it is reinitialized immediately with new positions and attributes.

As mentioned previously, the randomization of the particle values during recreated is performed using CURAND. A call to `curand_uniform` uses the current CURAND state to produce a random number in a uniform distribution between $0$ and $1$. Initializing the particles in a sphere requires the use of a polar coordinate system where the radius length and angles *theta* (azimuth angle in the $xy$ plane from the $x$-axis) and *phi* (polar angle from the $z$-axis) are randomized upon initialization. Simply randomizing latitude and longitude coordinates is not sufficient for a uniform distribution, as particles will be generated more densely at the poles of the sphere and more sparsely at the equator. Although a full polar-coordinate system using trigonometric functions is more computationally expensive, it is required to produce a completely uniform distribution of the particles inside the whole sphere. From here these po-

```
//Fire particle system update and re-initialization kernel
__global__ void fire_kernel(float4 *pos,float4 *vel,float4 *acc,float dt,
    float4 *col, curandState *state, float max_r, int numParticles){

    //Usual index calculation based on threads and blocks
    uint id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id >= numParticles)return;

    //Current life stored in position array, life decreased by decay
    pos[id].w -= (dt * vel[id].w);

    //Destroy and recreate new particle when needed
    if(pos[id].w <= 0.0){

        //Get the current CURAND local state
        curandState lstate = state[id];

        //Randomize initial values
        float theta = 2.0f * PI * curand_uniform(&lstate);
        float phi = acosf(2.0f * curand_uniform(&lstate)-1.0f);
        //Cube root of uniform fixes clustering in the center
        float radius = max_r * powf(curand_uniform(&lstate), 1.0f/3.0f);

        //Init starting position using polar coordinates
        pos[id].x = radius * sinf(theta) * cosf(phi);
        pos[id].y = radius * sinf(theta) * sinf(phi);
        pos[id].z = radius * cosf(theta);

        //Init velocity and acceleration to give flame like shape
        vel[id].x = 0.1f*(2.0f * curand_uniform(&lstate)-1.0f);
        vel[id].y = 1.0f + 0.5f*(2.0f * curand_uniform(&lstate)-1.0f);
        vel[id].z = 0.1f*(2.0f * curand_uniform(&lstate) - 1.0f);

        acc[id].x = 0.0f;
        acc[id].y = 0.3f + 0.1f * curand_uniform(&lstate);
        acc[id].z = 0.0f;

        //Calculate the rate of decay from $x^2 + z^2 = 0$
        float xz = make_float2(pos[id].x, pos[id].z);
        float len = length(xz);
        float decay = 0.5f + (0.5f * len / max_r);

        //Store max life and decay in pos/vel arrays
        pos[id].w = 1.0f;
        vel[id].w = decay;

        //Handle colour - outer flame = more red, less white/yellow
        col[id].x = 1.0f;
        col[id].y = 1.0f*(0.5f+0.3f*curand_uniform(&lstate))*(max_r-len);
        col[id].z = 1.0f*(0.0f+0.5f*curand_uniform(&lstate))*(max_r-len);
        col[id].w = 1.0f;

        //Update CURAND state
        state[id] = lstate;
    }
}
```

*Listing 6.1: Main kernel for a simple non-interacting particle model for fire simulation*

```
__global__ void fire_kernel(... float tw, bool wind ...){
    ...
    //Same creation/destruction process
    ...
    //Add wind component to the particles
    if(wind){
        acc[id].x = 3.0f * sin( ((2*PI)*tw)/0.5f );
        acc[id].z = 3.0f * sin( ((2*PI)*tw)/0.6f );
    }
    //Pass onto integration method
}
```

*Listing 6.2: Adding wind effects into the system*

lar coordinates are converted into cartesian coordinates using the standard conversion formula. This provides the final position of the particle when it is created.

Initial velocity and acceleration attributes are also randomized, depending on the type of effect which is desired from the simulation. The particles in a flame simulation rise upwards until they are destroyed and re-initialized in the emitter area - therefore the initial velocity should be set with a relatively high $y$-component to the velocity and acceleration, with lower $x$- and $z$-components. Some leeway may be given in these directions (the particles should sometimes deviate slightly from the intended direction) but this should be a minor deviation - the fire should not burn sideways.

However, if more turbulent behaviour is desired, an externally applied wind force can be applied uniformly across the system in the form of a simple sine wave function as shown in the general form as well as the implementation form in Equation 6.1. Listing 6.2 shows how the wind effects can be added to the main fire update kernel.

$$\begin{aligned} y(t) &= A \cdot \sin(2\pi f t) \\ a.x &= 3 \cdot \sin(4\pi t_w) \end{aligned} \tag{6.1}$$

In the general Equation 6.1, $A$ is the amplitude of the sine wave, $f$ is the wave frequency and $t$ is the current time. A phase shift in the sine wave is not needed unless the wind speed is intended to be particularly fast right at the start of the simulation, which is not required. For the implemention equation, the timestep $t_w$ has been named as such to distinguish it from the simulation timestep $dt$ because the wind force timestep is independent of the particle system. These values can be changed depending upon the type of wind desired; high amplitude and frequencies could represent violent changing winds, while low amplitudes and frequencies represent calmer winds. Additionally, this equation is only applied to the $x$-component of the particle acceleration, and the same kind of equation may be used to accelerate the particles in other directions as well. For example, in Listing 6.2 a second acceleration component is applied in the $z$-direction, but using a slightly different frequency so that the wind does not change equally in both directions. A sine wave representation of the wind force is obviously not an accurate real-world representation, but it is sufficient in this case to demonstrate the

ability to introduce turbulence into the fire particle system.

Visual results of the fire will be discussed in detail in Section 7.1. It will be shown that the non-interacting particle method is capable of producing a reasonable effect but it can be improved by firstly improving the method of governing the movement of the particles. For example, using the velocity-vortex method may provide an improved behavioural model for more realistic results. The implementation of the velocity-vortex method will be described in Section 6.4, but first a small detour must be taken to demonstrate the interaction between the particles and the spatial grid when applied to a simple case of a interaction-implementing particle system - such as "plasma" particle system.

## 6.3 Plasma Implementation

Before going over the implementation for the velocity-vortex method, it is important to adequately illustrate the difficulties in the coupling of the particle system and the spatial grid, as this has not yet been an issue when dealing with a non-interacting method. The description of a simplified plasma system is good for this purpose as it provides a simplistic representation of the problem of combining these two aspects when dealing with particles enacting forces over relatively long distances. It must be noted that the following implementation is not by any means a complete solution to a plasma physics simulation - the physics involved in plasma simulation are far more complicated than that described here.

Plasma can be defined as a state of matter where gas particles have been *ionized*. Solids, liquids, gases and plasmas make up the four fundamental states of matter in the universe, with plasma being by far the most abundant of the four. *Ionization* of particles occurs when extreme heat, pressure or electric discharge (i.e. a strong magnetic field) are applied to gas particles, changing the gas into plasma. These particles are called *ions* and can either be positively or negatively charged. The Aurora Borealis is also a plasma system, forming when *solar winds* - streams of charged particles from the sun - interact with the Earth's magnetic field. Closer to home, neon signs and plasma televisions both incorporate plasma systems. Plasmas are also central to the research of nuclear fusion and fusion energy [88]. The *quasi-neutrality* of plasma means that at a large scale the net charge of a plasma system is zero, meaning that the total number of positive and negative charges should be equal. This concept is important in relation to the simulation of plasma with particles; the plasma system can be assumed quasi-neutral because the particle system only models the plasma well on a macroscopic level.

Plasma simulation is often described as multi-scale or multi-level. This refers to the fact that plasma systems behave on a wide range of different lengths and time scales [99] [122]. For example, plasma can be simulated on a macro-scale, but using different theories compared to micro-scale simulations. This makes plasma systems difficult to simulate when trying to include all the relevant physics for each scale, so approximate models are used with trade-offs between accuracy and computational efficiency. A number of localization approximations are common place in molecular dynamical simulations [2]. Two main types of approaches to computational modelling of plasma systems are *fluid approaches* and *kinetic approaches*.

Fluid approaches such as hydrodynamics [68] or magnetohydrodynamics (MHD) [35] are the most popular. Kinetic approaches model plasma over larger ranges of density and temperature [37] and are more accurate, however they are more computationally expensive than fluid approaches [8]. One way of dealing with this problem is to use hybrid modelling techniques with elements of both fluid and particle approaches which makes compromises between computational effectiveness and accuracy [8].

The *Particle-in-Cell* approach (PIC) [123] [72] [79] is an example of a kinetic approach to plasma simulation. PIC methods work by dividing the simulation space up into cells, where particle-particle interactions *within* the cell are handled normally while particle-particle interactions *outside* the cell are handled based on the cell potential. Other well known kinetic methods include the Vlasov [36] and Fokker-Planck [100] methods. Kinetic modelling has been used in the simulation of interactions between plasma and pulse laser systems [63], and particle-based methods have been successful in simulating plasma systems relating to the motion of blood cells [131].

The calculation of particle fields and forces for plasma is complicated, and only a partial solution to the problem is presented in this thesis, specifically for illustrating the interaction with the spatial grid over long distances. It uses the the *Newton-Lorentz* equations of motion [11] to calculate the electrostatic and forces acting on the charged particles, however in this implementation electromagnetic forces are not applied. As such, this implementation resembles more of a *nbody* simulation [96] with multiple *species* of particles.

### 6.3.1 Plasma System Overview

The particle system used to simulate this plasma system will be a *state-preserving* particle system. The particles are persistent and are not destroyed after a certain time period. The rules of the plasma system are highly simplified for easier simulating and rendering purposes. The system can be described as a *multi-species* particle system similar to that of an *n-body* simulation [96] [52] [105]. Plasma particles have *charges*, either positively or negatively charged - therefore, a two species particle system, one species for each charge state. Positively charged particles will attract negatively charged particles, while at the same time repel other positively charged particles. Similarly, negatively charged particles will attract positive, and repel negative. The particles will be allowed to *pass through* each other - there are no body-body collisions between pairs of particles. However, if two particles are close enough, they will attract or repel each other depending on their relative charges. Thus particles will be continuously pushing away and pulling towards other particles. As mentioned earlier, it is assumed that the plasma system is *quasi-neutral* - random particles are assigned either a positive or negative charge, and the charge remains the same throughout the duration of the simulation.

The particles are initialized within a spherical region, in a uniform distribution with randomized polar co-ordinates using the CURAND random number generator (see Section 3.5.2). To maintain a balanced system, positively and negatively charged particles should be distributed *uniformly* throughout the system to ensure there are relatively equal forces acting on

the particles in the system. Particle velocities and accelerations are initialized to zero. Particle positions, velocities, accelerations and charges are stored in arrays in *global* memory.

Particle properties are initialized in *constant* memory. The system has been simplified when dealing with the particle's *mass* and *temperature* properties. It is assumed that all particles are of the same mass, while it is also assumed that all particles' temperatures are constant (the system is *isothermal*).

Plasma physics can most easily be modeled using an electrostatic model [11], where particles are charged due to their own electrostatic fields. In an electrostatic model, particles exert an *electrostatic force* on each other. In general, the force acting on a charged particle is given by Lorentz' force equation, as shown in Equation 6.2.

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \tag{6.2}$$

where $q$ is the individual particle charge, $\vec{E}$ is the electric field, $\vec{v}$ is the particle velocity and $\vec{B}$ is the magnetic field. To simplify the equation, the force exerted on the particle can be made up of two parts, as shown in Equation 6.3 [11].

$$\begin{aligned} \vec{F} &= q\vec{E} + q(\vec{v} \times \vec{B}) \\ \vec{F} &= \vec{F}_{electric} + \vec{F}_{magnetic} \end{aligned} \tag{6.3}$$

Here is where the implementation differs from the general plasma physics. In this implementation, the magnetic field force is not implemented - instead, the interactions between pairs of particles is based directly on the electrostatic element. This is the force which directly draws the pairs of particles together - Gauss's law for magnetism states that the magnetic field "force" is simply a rotation of the velocity vector [11].

As Equation 6.3 states, $\vec{F}_{electric} = q\vec{E}$. The electric component of the force is given by Coulomb's law [24], which gives the magnitude of the electrostatic force between two point charges $q_i$ and $q_j$ as shown in Equation 6.4.

$$\vec{F}_{i,j} = \frac{1}{4\pi\epsilon_0} \frac{|q_i q_j|}{r_{i,j}^2} \tag{6.4}$$

where $r_{i,j}$ is the scalar distance between the two particles. This distance can be calculated based on each particles position vectors. The vector form of the equation is calculated by multiplying by the distance vector $r$ as shown in Equation 6.5. Additionally, the term $\frac{1}{4\pi\epsilon_0}$ is known as *Coulomb's constant* and can be written as $k_e$.It is notable that this electrostatic force is an inverse-square force calculation similar to other equations in this thesis. For example, the force due to gravity (Equation 4.5) and Biot-Savart's Law for velocity due to vorticity (Equation 4.20) both follow this pattern.

$$\vec{F_{i,j}} = k_e \frac{q_i q_j}{r_{i,j}^3} \vec{r} \tag{6.5}$$

Note that $r_{i,j}$ refers to the scalar distance between particles $i$ and $j$, while $\vec{F_{i,j}}$ refers to the force acting on particle $i$ from particle $j$ and $\vec{r}$ refers to the relative position vector between the two particles (from $i$ to $j$). This force will subsequently be used by the numerical integration method to determine the final motion of the particle.

### 6.3.2 Plasma-specific Spatial Grid Analysis

It is mentioned earlier that a different approach to calculating interaction forces needs to be made when dealing with the electrostatic attraction and repulsion forces inherent in the particle system. The original setup of the spatial grid is designed to account for sphere-to-sphere collisions - physical contact between bodies. Electrostatic field attraction and/or repulsion forces influence particles at a much larger distance than body-body collisions, so this traditional method does not work. As a reminder, the application of the equation for electrostatic force generated between two point charges is given earlier by Equation 6.5.

Two different issues arise when considering applying an electrostatic force to a particle in a spatial grid:

- Many particles intending to be attracted/repulsed are likely not to be within the 3x3x3 neighbouring grid cells set up for the standard body-body collisions. Ignoring the particles outside of such a small distance would result in very little action in the system as a whole.
- The force exerted by each particle's electrostatic field would essentially reach an infinite distance, because as shown by Equation 6.5 the force is simply a function of the distance between the two bodies. Exerting a force on every other particle defeats the purpose of the spatial grid to begin with, and the computational complexity returns to an order of $N^2$.

Solving these issues involves compromising between the number of particles that *can* be influenced by the electrostatic force - i.e. the spatial grid - and the number of particles that *will* be influenced - i.e. the size of the particle's electrostatic field. More explicitly, two modifications to the particle system will need to be made:

- The number of inherent particle-to-particle interactions needs to be increased, but not so much that all particles influence each other. This can be done in two ways - either increase the number of neighbouring cells to check for interactions, or increase the size of the spatial grid cells (grid maximum/minimum values remain the same). Doing this will increase the total number of particles that may be influenced by the electrostatic field, but still keep it within a local area. The second option is the one used in this thesis but both options essentially do the same thing.

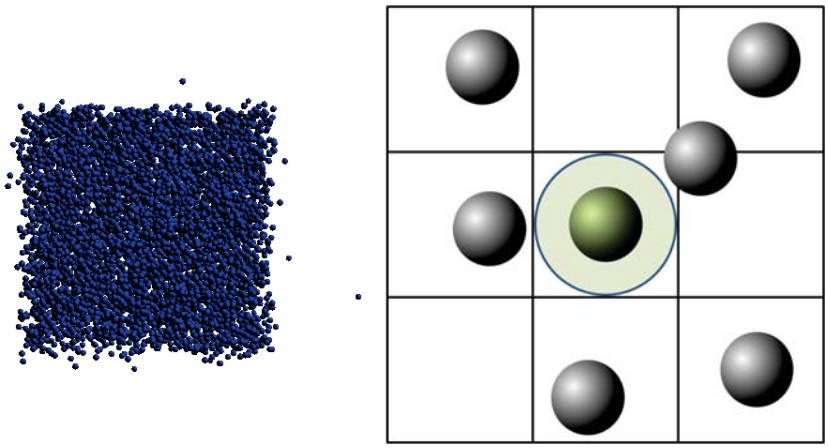

*Figure 6.1: The electrostatic field size is not large enough to reach many neighbouring particles, even though some particles are contained in neighbouring cells in the spatial grid. Only particles which are very close to the particle in question are affected. This causes very little change from the initial state (as shown on the left, the particles are initialized randomly in a cube structure) until particles have moved such that all fields have no effect and damping forces cause the system to come to a standstill.*

- Apply some limit on the force exerted by the particle's electrostatic field. This involves setting a "cutoff" distance, after which particles at a distance larger than this value will be assumed to have zero force exerted on it by the current particle's field.

Combining both these points allows the spatial grid to continue to be used effectively to reduce the total number of particle-particle interactions and speed up the simulation, although it will inevitably be slower than it would using only body-body collisions because there will always be a greater number of interactions to consider.

Figures 6.1, 6.2, and 6.3 show different scenarios that can occur when trying to balance the size of the particle electrostatic field and the size of the spatial grid cells. Each figure contains both a screenshot of the particle system and an explanatory diagram showing the general set up of the system. The particle systems in the screenshots are set up specially to demonstrate the interactions between the particles in the simplest possible way - all particles' charges have been set to be identical. Thus it is expected that the particles all repel each other and should disperse in a relatively uniform manner.

Particles are initialized in a cube structure (to clearly demonstrate the level of dispersion) and are contained in an invisible bounding sphere. Additionally, all systems are initialized using the same random seed and screenshots are all taken after 1000 iterations of the simulation. Keep in mind that while the explanatory diagrams are drawn for a two-dimensional grid, this is simply for ease in demonstrating purposes. In the case of the plasma particle system it actually applies to a three-dimensional spatial grid rather than a two-dimensional one.

The behaviour in Figure 6.1 occurs when the particle's electrostatic field does not influence enough particles. Even though there are particles within neighbouring cells and these particles are checked for collisions, they are ultimately ignored because the cut-off radius is too small.

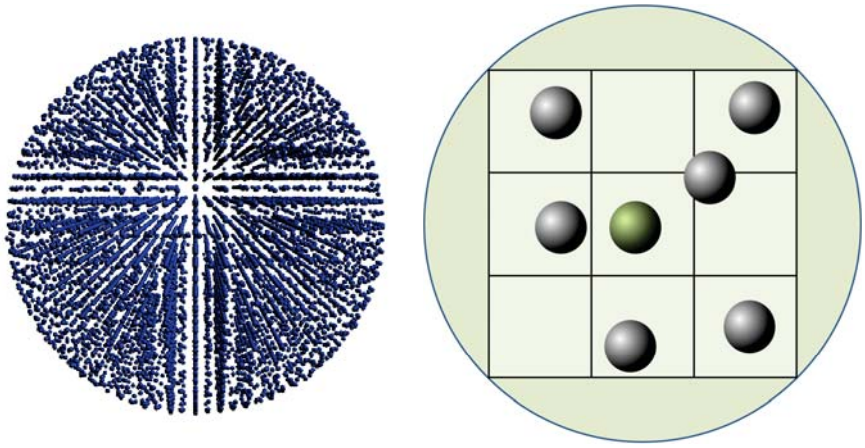

*Figure 6.2: Allowing the field size to become too large causes too many particles to be influenced by any one field, arranging the particles in a uniform pattern corresponding to the arrangement of the spatial grid cells. Since the particles are still assigned to various spatial grid cells, only those within the current cell and neighbouring cells are checked and the particles which are not contained in neighbouring cells, though still within the field, are incorrectly ignored.*

Because of this, too few particles attract and repel each other until a state is reached where no two particles are within the influence radius of each other. This is demonstrated clearly by the fact that the system has remained largely in a cube-like state and very few particles have dispersed as intended.

The behaviour in Figure 6.2 occurs when the particle's electrostatic field is too large in relation to the spatial grid's cell size. In this case, every particle within the current and neighbouring cells is contained within the field radius, while particles in cells outside of the local environment are not checked despite being within or partially within the field. The system in Figure 6.2 has been contained within a bounding sphere in order to demonstrate the effect caused by not processing interactions with these outer particles.

Figure 6.3 shows the system set up correctly, with all particles dispersing evenly and no particles within the field influence radius being ignored. In this particular particle system multiple particles may move around within each others electrostatic fields. The rule for setting up the spatial grid becomes that the size of the spatial grid cell the size of the *field influence radius* rather than the particle's radius. This ultimately amounts to more particle-particle interactions, as there are multiple particles within each field which will be drawn to each other. Again, the particles are contained in a bounding sphere to demonstrate this effect.

Listing 6.3 shows this revised approach to the force calculation step of the simulation. Using a pre-defined field size constant, the electrostatic force is calculated using the formula in Equation 6.5 only for particles which are sufficiently close to the particle in question.

Note that there has been modification to the distance formula in order to avoid singularity instances - i.e. when particles both occupy exactly the same space, the resulting force is extremely high due to the distance between them being extremely low. The $soften$ variable

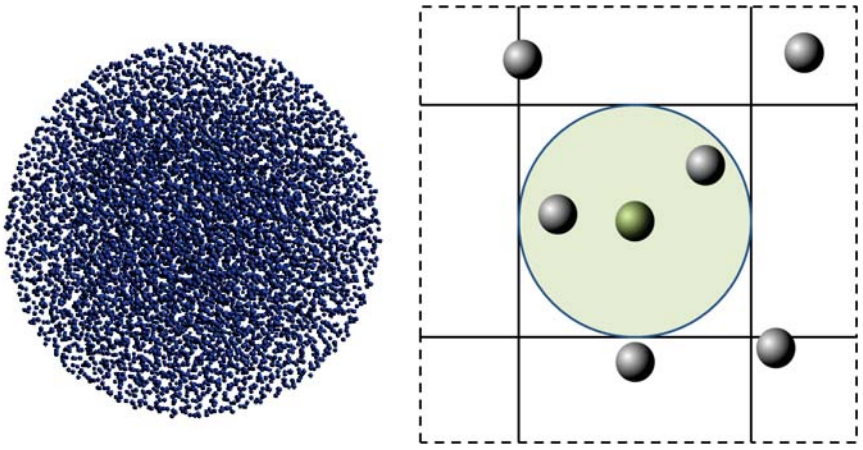

*Figure 6.3: The size of the electrostatic field is appropriate in relation to the size of the spatial grid cell. Particles are able to interact with each other correctly and exert repulsive forces on each other so that they spread out evenly within the sphere.*

ensures that pairs of particles have a minimal distance between them to avoid these extreme forces causing instability in the system. Once the force has been accumulated for all particles within the field radius, it is passed along to the particle integration method.

Figure 6.4 shows the execution times of the kernel which calculates the electrostatic force exerted on each particle, where the cell size of the spatial grid has been changed to accommodate more or less particles in each cell and subsequently the electrostatic field has also been altered to match the layout shown in Figure 6.3.

The size of the particle system used when obtaining these times remained at a modest 32768 ($2^{15}$) particles. The first data point represents the cell size for the spatial grid described in Section 5.1.3 designed for body-to-body collisions; i.e. the field extends only so far as the particle's own collision radius. Subsequent data points are sampled using multiples of this initial field size. Steadily increasing the particle influence radius results in more particles per cell (and subsequently neighbouring cells) to check for interactions, the execution time increasing somewhat exponentially as the size increases.

Towards the top end of the graph, the execution times begin to level off, as the cell sizes become too large to be contained in the grid. In a grid with dimensions 128x128x128 (as the one used in this graph), a cell size multiplier of $2^5$ results in only 64 cells total, 27 of which will be checked for interactions between pairs of particles in the current and neighbouring spatial grid cells. At a cell size multiplier of $2^6$, the grid space is divided up into only 8 cells, and checked for particle interactions as all cells will be considered one of each other's 27 neighbouring cells. At this point, the usage of the spatial grid has become impractical.

Given the execution times shown in the graph, it would be reasonable from a computational perspective to use a spatial cell size multiplier of around $2^3$, at least when using a particle system of this particular size. This value ensures a reasonable influence radius to calculate the electrostatic forces of the particles, while sacrificing very little execution time when compared to the standard body-to-body spatial grid. The behaviour of the system also needs to be taken

```
__device__ float3 plasma_electrostatic(float3 posI, float3 posJ,
    float chargeI, float chargeJ, float rI, float field_size,
    float k_coulombs, float soften){

    //Calculate the relative position
    float3 rel_pos = posJ - posI;
    float3 force = make_float3(0.0f);

    //Calculate the electrostatic field radius (influence)
    float dist = length(rel_pos) + soften;

    float dist_inv = 1.0f /dist;
    dist_inv *= dist_inv * dist_inv;            //r^3
    float influence = rI * field_size;

    //Only act on particles within influence radius
    if(dist < influence){

        float f = k_coulombs * (-(chargeI.x * chargeJ.x)) * dist_inv;
        force += f * rel_pos; //get force vector
    }

    //Return the final force
    return force;
}
```

*Listing 6.3: The calculation of electrostatic attraction and repulsion forces acting on particle i from particle j*

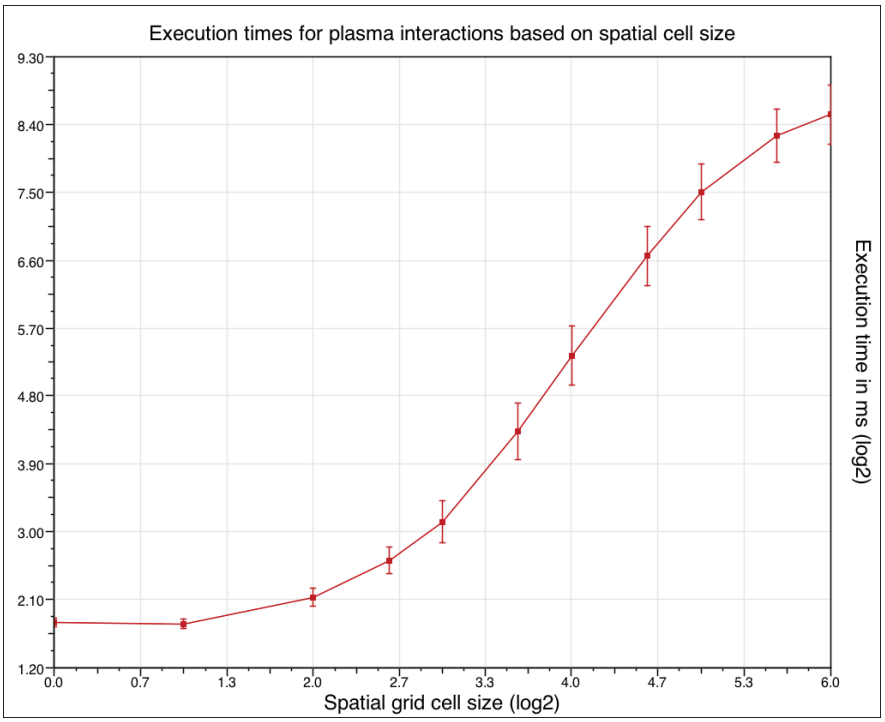


*Figure 6.4: Execution times of the force calculation kernel given increasing sizes of the spatial cell and electrostatic influences.*

into account when deciding the size of the influence radius - the more particles are influenced, the more accurate the simulation will be, but it will run slower. Therefore a compromise can be arranged, depending on the desired behaviour of the particle system and/or the level of potential from the GPU hardware.

### 6.3.3 Interaction Processing

Once the grid cell size has been calculated, the spatial grid can now begin in helping process interactions. The position of the cell each particle resides in is calculated and then a list of the other 26 neighbouring cells is generated. These neighbours should include not only the six nearest-neighbour cells in both positive and negative directions on each axis ($x$, $y$ and $z$), but also the cells neighbouring diagonally, as it is possible that particles classified with these cell indices could also partially lie within the current cell. Once this list has been calculated, interactions are performed between the current particle and every particle classified as having the same cell index, and this is repeated for each neighbouring cell. These indices into the particle arrays with which the current particle needs to collide are found using the `first` and `last` arrays created by the spatial grid. This method is demonstrated in Listing 6.4.

The interaction method assumes both interacting particles have the same radius and mass, and both are assumed as constant throughout the simulation. Charge is applicable when calculating the electrostatic force acting on the particles (see Listing 6.3. In addition to particle-particle interactions, additional forces such as interaction with other objects or boundaries must also be taken into account. This is performed after the particle-particle force calculation, as shown in Listing 6.4 but the details of which are omitted from the listing as they do not affect the overall collision method and are simply situational depending on the simulation.

Once the total force for each particle has been accumulated, including particle-particle interactions between particles in the current cell and neighbouring cells, as well as objects or boundaries, the force is passed along to the system's integration method as the acceleration component.

The implementation of this plasma system is a good demonstration of the challenges involved in coupling the particle system and the spatial grid over large interaction distances. This will become relevant later on, especially in rendering surfaces using field sizes reaching across multiple particles. See Chapter 7 for more information on this. Next, the velocity-vortex method for fire is implemented.

## 6.4 Velocity-Vortex Fire Implementation

The fire velocity-vortex implementation has several additional elements which improve on the basic non-interacting particle model.

- *Spatial Grid:* Particle-to-particle interactions must be processed using this model, so the same spatial grid implementation used for the water and plasma simulations is used here as well. The grid will serve double as the grid for the Eulerian-specific attributes of

```
__global__ void calc_electrostatic(float4* output_force,float4*sorted_pos,
    float4*sorted_vel,float*sorted_charge,uint* cell_first,uint*cell_last,
    float r, float field, float k_coulombs, float k_singular){
    ...
    //Get particles data from sorted arrays
    float3 pos = make_float3(sorted_pos[index]);
    float3 vel = make_float3(sorted_vel[index]);
    float charge = sorted_charge[index];

    //Get address in the grid
    int3 cell = calcCellPos(pos);
    float3 force = make_float3(0.0f);

    //Examine neighbouring cells
    for(int z=-1; z<=1; z++){
       for(int y=-1; y<=1; y++){
          for(int x=-1; x<=1; x++){

             //Neighbour is current cell when x,y,z = 0
             int3 neighbour = cell + make_int3(x, y, z);
             uint cell_index = calcCellIndex(neighbour);
             uint first_index = cell_first[cell_index];

             //Check the cell is not empty
             if(first_index != 0xffffffff){

                //Find last particle id in this cell
                //Now we have full list of particles in this cell
                uint last_index = cell_last[cell_index];
                for(uint i=first_index; i<last_index; i++){

                   //Check it is not colliding with itself
                   if(i != index){

                      float3 pos2 = make_float3(sorted_pos[i]);
                      float3 vel2 = make_float3(sorted_vel[i]);
                      float charge2 = sorted_charge[i];

                      //Apply electrostatic force
                      force += plasma_electrostatic(pos, pos2,
                         charge, charge2, r,
                         field_size, k_coulombs, k_singular);
                   }
                }
             }
          }
       }
    }
    //Output final force calculation
    output_force = make_float4(force, 0.0f);
}
```

*Listing 6.4: Interacting particles within current and neighbouring cells, applying electrostatic forces.*

the velocity-vortex model - cell velocity, cell density and the velocity Jacobian matrix. Unlike the water particle system, the fire particle system is not contained in a container - rather, the particles are free to move anywhere they want. This requires the spatial grid to be *dynamic*, scaling with the system as it grows and shrinks.

- *Cell data:* Velocity is stored not only for every particle, but also for each cell in the cell grid. This is intended to provide the functionality that particles "follow" the natural flow of the fluid, whereby the velocity for each particle is interpolated over the surrounding grid cells and then used as the velocity for the next integration step. Cell density also plays a role in determining the vorticity of each particle in the system.

- *Advection:* As there are Eulerian fluid dynamics involved in this implementation, an advection step must be provided - namely, the cell velocity is advected by the particle's vorticity. While position uses a standard integration step, the velocity used depends on the cell velocity, which is ultimately determined by the diffusion of the vorticity of all particles in the cell and neighbouring cells. This step was also tested globally (i.e. all particles influence each cell).

- *Body interaction:* Particles spawn around and interact with a static body, inducing a vorticity on the particles as they rise. As the vorticity of each particle is zeroed upon initialization, the induced vorticity by the body is the catalyst for the fluid-like behaviour of the particles in the simulation. For the purpose of this simulation, a simple spherical body is sufficient to provide the desired visual effect, although technically any shaped body may be used; a good example would be lighting a match - in this case a prolate spheroid would work very well as the approximation of the match head.

- *Vorticity diffusion:* Vorticity is propagated throughout the fluid particles by means of vorticity diffusion - neighbouring particles share their vorticity. In practice, this allows some particles which interact with the body to pull along neighbouring particles which otherwise would not have had an interaction. Additionally the amount diffused is moderated by the fluid viscosity, so that fluids with very high viscosity diffuse a lot more vorticity than low viscosity fluids - large vorticity diffusion means particles follow their neighbours much more closely.

- *Density calculation:* Vorticity due to density in the grid cell is also calculated, meaning that particles rotate towards areas of low density in the fluid. Density is also used when rendering the simulation, which will be explained later in Section 7.3.

There an important thing to note about the implementation of this model; it still makes use of the creation and destruction phase critical to a fire simulation. A fire simulation *must* have this phase as there is no way for particles to move from the top to the bottom in a circular motion without providing a poor visual effect and even fluid-like behaviour (fire does not move in a ring-motion, it rises straight upwards). It is also impossible to simply continue creating new particles indefinitely, although this would not solve the inherent problem anyway. This

problem is that the fluid dynamics equations are simply not designed to be used in tandem with this creation-destruction mechanic. The sudden injection of a particle at a particular point (the emitter) and the sudden removal of a particle from a particular area essentially means that the conservation of physical attributes of the fluid (such as momentum) in these particular areas is very difficult and can easily lead to instability. This is not a problem in the SPH implementation of the water because the particles in the system are never destroyed and recreated.

### 6.4.1 Fire System Initialization

To initialize the system, memory must be assigned to the graphics card for multiple additional particle and spatial grid elements as discussed in the previous section. Memory for the vorticity is assigned for each particle in the system - this can effectively replace the particle's acceleration attribute as there are no other forces applied to change the velocity other than the particle's vorticity. Separate memory must be allocated for the vorticity diffusion calculations, as vorticity data must not be modified by other threads in the system when diffusing the vorticity between two particles. Cell density and cell velocity are also assigned to each cell in the spatial grid. A viscosity value is set for the entire fluid which remains constant throughout the entirety of the simulation. As a fire is a gaseous substance it can be considered in having relatively low viscosity - for this simulation, a viscosity value of $0.05$ produced some reasonable results.

The most efficient set up for the spatial grid relative to the number of particles is to try and keep the number of spatial grid cells close to the number of particles in the system. In practice, the spatial grid will increase to at least twice the initial size as the particles rise up from the initial rest position on the rigid-body emitter. However, the size of the individual grid cells should remain the same and subsequently the influence radius of the particles (vorton radius) should be set at half the size of the grid cell dimensions - grid cells should be cube shaped for the vortex physics to work properly. Once the system has been initialized, the first thing required is to update the spatial grid dimensions dynamically.

### 6.4.2 Dynamic Spatial Grid

A dynamic spatial grid is required for the fire particle system as the particles are not restricted to a container with set dimensions in a static position. Fire particles simply rise from an emitter body and go where they please. In practice, for a standard simulation case the fire particle system will come to some equilibrium size, where particles will very rarely rise higher than the average lifespan will allow them to and subsequently recreate them back at the emitter body. In this case, a static spatial grid would be appropriate once it reaches this equilibrium state, but before this time - such as when the particles initially rise - it would not be as efficient. Additionally, a static spatial grid cannot account for behaviour in the fire system when external wind forces are applied to the system. The grid needs to account for particles which are blown in the far $x$- and $z$-extremities in this case, or else these particles will be classified as outside the grid and will be unable to interact with other particles in a proper fluid-like manner.

```
//Atomic max of particle position for dynamic spatial grid
__device__ inline float fAtomicMax(float* address, float pos){

    int* address_as_i = (int*)address;
    int old = *address_as_i, assumed;
    do{
        assumed = old;
        old = atomicCAS(address_as_i, assumed,
                __float_as_int(fmaxf(pos, __int_as_float(assumed))));
    }while(assumed != old);
    return __int_as_float(old);
}
```

*Listing 6.5: Using the atomic compare-and-swap to calculate the maximum value for the positions of all particles. This is done for each of the $x$- $y$- and $z$- elements of the particle's position vector. Additionally, a second implementation of atomicMin is used, which differs only in that the function $fminf$ is used as the comparison for the atomicCAS instead of $fmaxf$. The code compares and swaps the position value in parallel if it is larger than the previous value, and keeps doing this until maximum value has not been modified by any other thread.*

The dynamic spatial grid used in this simulation is a simple implementation where additional spatial grid cells are added or removed as the system grows or shrinks. The size of the spatial grid cell is set based on the vorton radius and remains static for the entire simulation, while the dimensions of the grid change over time. Determining the dimensions of the grid is done by calculating the maximum and minimum values of the positions of all particles in the $x$-, $y$- and $z$-dimensions. The total difference between the maximum and minimum values for each direction makes up the entire dimensions for the spatial grid. For each $x$-, $y$- and $z$-dimension in the grid, it is divided by the corresponding grid cell size calculated earlier to obtain the number of cells in each dimension in the grid. This will change dynamically over time as the particles in the system move.

Calculating the maximum and minimum values for all particle positions can be very costly performance-wise for large particle systems. For this reason a parallel method is implemented to speed up this process, using the $atomicMax$ and $atomicMin$ functions built into CUDA. However, these typically do not work with floating point numbers (positions are stored as floating point values) so a work around must be used using the atomic compare-and-swap as described in the CUDA Toolkit Documentation [91]. This workaround is shown in Listing 6.5.

In practice, the atomic maximum and minimum functions should not need to compare very many times, as the number of particles which increase or decrease the maximum or minimum values each timestep will be very few. However, this is only the case if the spatial grid is only allowed to increase the dimension size and not decrease it. In order for the grid to decrease as well, the maximum and minimum values need to be reset and the comparison would be performed many more times on average. In the case of a steady flame simulation, the first case would be fine to implement as the grid dimensions would not decrease in size beyond some equilibrium size, as discussed earlier. However, if wind effects are introduced, the spatial grid must be able to increase and decrease its maximum and minimum values, because as the flame

is blown left and right, backwards and forwards, the maximum values in those directions must decrease opposite to the direction of the wind, and vice versa for the minimum values. If this does not occur, the spatial grid becomes unnecessarily large relative to the number of particles in the system, decreasing the effectiveness of the spatial indexing and affecting the vorticity diffusion between cells at the edges of the flame.

### 6.4.3 Vorticity Dynamics

Once the spatial grid has been dynamically adjusted based on the current positions of the particles in the system, the velocity values for each cell in the grid can be calculated. The spatial grid is referred to as the *velocity grid* when talking about the velocity values. This step is parallelized using one thread per *grid cell* rather than the standard one thread per particle.

Velocity is calculated based on the vorticity of the particles contained in each grid cell. This step assumes the sorted arrays for particle indices have already been calculated. Given this assumption, the cell's position in the grid is calculated using the cell's own index number, which is then used to generate the list of possible neighbouring cells. Each of these cells (including the current cell) provide indices into the *cell_first* and *cell_last* arrays containing the sorted indices of particles contained in those cells. Altogether this provides a complete list of particles which will add velocity to the current cell. If the current cell is on a boundary in any direction, the neighbouring cells *in the same direction* should not provide any additional velocity, as the spatial grid *does not wrap around on itself*. In this case a simple check is performed - if the cell is a boundary cell, the velocity accumulation for all neighbours in that particular direction is set to zero. Algorithm 6 shows this method in detail.

In the algorithm, the function $velFromVort$ is simplified and is shown in greater detail in Listing 6.6. As explained in Section 4.4, velocity is due to vorticity is given using Biot-Savart's law (See Equation 4.20). Notice that the velocity calculated uses the particles angular velocity, which is simply half the vorticity. One thing to note is that a check occurs in relation to the position of the vorton relative to the cell position. If it is very close, it is not desirable to induce a huge velocity due to the vorticity, so a corrected velocity is calculated.

As explained in Section 6.4, the change in vorticity (or vortex stretching) every timestep is dependent on the velocity of the fluid around the vorton. This step involves two main parts - firstly, the computation of the Jacobian matrix of the velocity in the cell in which the vorton resides, and then secondly applying the stress formula to find the new vorticity. Calculating the Jacobian is fairly simple and uses the formula already given in Equation 4.21. The computation of the Jacobian velocity is parallelized one velocity grid cell per thread and outputs the Jacobian matrix for each cell to be used by the next part.

The more important part of this step is the stretching (change) of the vorticity using the stress formula, referred to previously as the stress term in Equation 4.19 and subsequently elaborated as the matrix of all partial derivatives of the velocity (the Jacobian) in Equation 4.21. Once the Jacobian matrix of the velocity has been calculated, the velocity due to vorticity kernel is executed parallelized at one thread per particle, taking the Jacobian matrix as an input. An

**Algorithm 6** In-depth velocity calculation due to particle vorticity for velocity grid

```
Given output array U of c cell velocities
Given array P of n particle positions
Given array V of n particle vorticities
Given arrays F and L of sorted particle indices (First and Last)
Given vorton influence radius R
Given spatial grid values - minimum (origin) O, cell size S
//Spatial grid arrays have already been sorted
//Cell position is a function of its own index number
//Cell world position is a function of spatial grid values
for each grid cell in PARALLEL, with thread index i do
  Cell C = cellfunc(i)
  Cell world position W = wFunc(C, O, S)
  //Find neighbours. Neighbouring cells are -1 to +1 in all directions
  for all -1 ≤ x ≤ +1, -1 ≤ y ≤ +1, -1 ≤ z ≤ +1 do
    Neighbour N = C + Vec(x, y, z)
    //If neighbour outside boundary, ignore
    //Otherwise, obtain list of neighbouring particle ids in F and L
    if insideBoundary(N) then
      cell_index = cellfunc(N)
      f = F[cell_index]; l = L[cell_index]
      //Apply velocity due to vorticity for all neighbouring particles
      for p = f; p ≤ l; p++ do
        U[i] += velFromVort(P[p], V[p]/2, W, R)
      end for
    end if
  end for
  New cell velocity has been calculated. This is passed to the next step.
end for
```

overview of the stretching step is shown in Algorithm 7. It involves calculating the velocity grid cell based on the particle position which is fairly standard, as well as the cell's world position. These two positions are used to calculate the spacing during linear interpolation. Next, a linear interpolation of the given Jacobian matrices is performed, finding the final interpolated matrix $lerpJ$. This matrix is used to transform the vorticity vector into the new *stretched* vorticity. This is multiplied by the given timestep to arrive at the final vorticity.

The next step in the algorithm is the diffusion of vorticity - basically, nearby vortons share vorticity amongst themselves. This step follows the implementation of the viscous term of the Navier-stokes equations - it applies the fluid viscosity to the diffusion of the vortices as described earlier in Equation 4.22. There are two possible ways to go about this. Firstly, the vorticity can be diffused globally throughout the entire fluid. This means every vorton shares its vorticity with every other vorton - there is clearly a problem with this method however, as it does not scale well to larger system sizes.

A better method is to diffuse vorticity only in the local area. Because the amount of vorticity diffused is dependent on how close the vortons are to each other, a cutoff distance can be

```
//Calculate cell velocity due to vorton vorticity
__device__ float3 vel_from_vort(float3 cell_pos, float3 vorton_pos,
                                float3 ang_vel, float radius){

    float3 rel_pos = cell_pos - vorton_pos;
    float dist = length(rel_pos);
    float radius_sq = radius * radius;
    float dist_sq = dist * dist;
    float effective_dist = 0.0f;

    //Handle case where vorton is very close to cell pos
    //i.e make sure it doesn't induce a huge velocity (singularity)
    if(dist_sq < radius_sq) //inside vorton
        effective_dist = 1.0f / (radius * radius_sq);
    else                    //outside vorton
        effective_dist = 1.0f / (dist * dist_sq);

    //Cross product w x r
    float3 cp = make_float3(
        ang_vel.y * rel_pos.z - ang_vel.z * rel_pos.y,
        ang_vel.z * rel_pos.x - ang_vel.x * rel_pos.z,
        ang_vel.x * rel_pos.y - ang_vel.y * rel_pos.x
     );
     // (w x r) / (4 * Pi * r^3)
     return (1.0f / FOURPI) * cp * effective_dist;
}
```

*Listing 6.6: Calculating the velocity of a grid cell due to vorton vorticity, using Biot-Savart's law in relation to the angular velocity of a vorton. Notice here that an extra step has been taken to limit the velocity induced when the cell position is inside the vortex radius. In this case it restricts the effective distance r to the vortex radius, rather than the actual distance between the two points.*

set whereby it is assumed that the amount of vorticity diffused is so small that it is ignored completely. Over several iterations this vorticity will propagate through the fluid as vortons continue to move around. As mentioned earlier, this method is known as the Particle Strength Exchange (PSE) method [140]. The local vorticity diffusion method is much more preferable in this case for larger system sizes and it can easily take advantage of the spatial indexing already provided by the spatial grid (velocity grid).

The diffusion step looks very similar to the velocity from vorticity step. Firstly, the cell ID is calculated based on the vorton position and the list of neighbouring cells are calculated based on this cell's position in the grid. Then, for each of these neighbours, indices into the first and last sorted arrays are calculated to obtain a list of all vortons with which the current vorton will diffuse its vorticity.

Similar to the velocity from vorticity step, each neighbouring cell is checked to see whether it is outside the boundary of the velocity grid - if so, it is ignored and no vorticity is diffused to that cell. Otherwise, diffusion occurs for each pair of vortons by multiplying the viscosity of the fluid by the difference in vorticity of the two vortons in question.

Since this is happening all in parallel, the amount of vorticity diffused per vorton is doubled - this handles outgoing vorticity from the current vorton and incoming vorticity from the partner

**Algorithm 7** In-depth vorticity due to cell velocity algorithm for vortons.

```
Given array V of n particle vorticities
Given array X of n particle positions
Given array J of c Jacobian matrices for cell velocity
Given the advection timestep t
Given spatial grid values - minimum (origin) O, cell size S
//Spatial grid arrays have already been sorted
//Cell position is a function of particle position
//Cell world position is a function of spatial grid values
for each particle in PARALLEL, with thread index i do
    Cell C = func(X[i])
    Cell world position W = wFunc(C, O, S)
    //Perform a trilinear interpolation over the surrounding Jacobians
    //Let each corner be considered a "neighbour"
    Given Neighbours N[8]
    for all a in 8 corners of the cell cube do
        Corner = Vec(x, y, z) where x, y, z are positive 0 or 1
        N[a] = C + Corner
    end for
    Mat3 lerpJ = Lerp(X[i], W, N, J)
    //Given lerpJ, find dot product of angular velocity
    Vec3 stretch = V[i] * lerpJ
    V[i] = stretch * t
end for
New particle vorticity has been calculated. This is used in the next step.
```

**Algorithm 8** Algorithm for the diffusion of vorticity in the local space.

```
Given output array D of diffused vorticities
Given array N of neighbouring vortons between F and L
Given array X of n vorton positions
Given array V of n vorton vorticities
Given mu, the fluid viscosity
Given the advection timestep t
//Spatial grid arrays have already been sorted
//Cell position is a function of particle position
for each vorton in PARALLEL, with thread index i do
    D[i] = V[i]
    Cell C = func(X[i])
    //Diffuse between all neighbouring vortons
    for all a in N do
        Vec3 vortdiff = V[i] - V[a]
        Vec3 shared = vortdiff * mu * t
        //Apply cutoff radius
        if X[i] - X[a] < cutoff_radius then
            D[i] -= shared * 2
        end if
    end for
end for
Diffused vorticity has been calculated. This is used in the next step.
```

**Algorithm 9** Buoy fire particles so they rise upwards

```
Given array U of n vorton velocities
Given desired velocity B
Given buoy co-efficient C
//For each vorton, make it move towards the desired velocity
for each vorton in PARALLEL, with thread index i do
    Vec3 veldiff = B - U[i]
    Vec3 buoy = C * veldiff
    U[i] += buoy
end for
```

vorton (outgoing vorticity is subtractive, while incoming vorticity is additive). Algorithm 8 gives a brief overview of this method.

### 6.4.4 Buoyancy Handling

The handling of buoyancy forces is specialized specifically for the fire system. In this case, it refers to the tendency for particles to rise upwards. Buoyancy can be applied in many ways, however this simulation uses a simply buoyancy calculation applied directly to the particles' velocity. Given a desired velocity, the difference between the current velocity and the desired velocity is calculated and is then increased or decreased depending on a given buoyancy coefficient. The desired velocity is implemented as a constant value in the simulation, and in the case of the fire this velocity should be moving upwards. This buoyancy calculation is effectively the same as the calculation of pressure forces due to fluid density as used in the Smooth Particle Hydrodynamics simulation for water. Algorithm 9 provides a general overview for this step, although it is fairly simple to implement and easy to understand.

### 6.4.5 Advection Step and Body Collision

The advection step of the vortex-velocity method works similar to the integration method in the other particle system implementations, but it is different in that the velocity of the particle *follows the flow* rather than use its own independent velocity. For it to follow the flow, the particle's velocity is taken from the current grid and surrounding grid cells using a trilinear interpolation, similar to the calculation of the Jacobian matrix for velocity. Interpolating over many cells in the grid means that the particles will move along with and through the fluid smoothly and not be affected by sudden changes in the direction of the velocity from grid cell to grid cell. Once the velocity for the particle has been found, it is used to step the position forward as per the usual integration method.

Body collision is handled after the advection step has been completed. As mentioned previously, the body used for this simulation is a standard sphere around which particles are initialized and begin to rise up due to the buoyancy force. The interaction of the particles with the body comes in two steps: firstly, any particles moving inside the body are pushed out, and secondly, moving around the body updates the velocity and vorticity of the particle.

**Algorithm 10** Interaction of vortex particles with the rigid-body emitter.

```
Given array V of n vorton vorticities
Given array U of n vorton velocities
Given array X of n vorton positions
Given the vorton effective radius R
Given body position B and radius BR
//First, check if the vorton interacts with body
//Vorton interacts with body if distance between them is smaller than the sum of the radii
for each vorton in PARALLEL, with thread index i do
  if X[i] - B < (BR + R) then
    Calculate point of contact C on the body
    Calculate velocity UC at contact point using standard Biot-Savarts law
    //Vorton is inside the body, needs to push out
    Calculate normal N between body and vorton
    Push vorton back along N until C
    Update new position X[i] use to UC
    Update new vorticity V[i] due to UC at contact point C using vorticity from velocity
    calculation
  end if
end for
```

The overall steps are shown in more detail in Algorithm 10. Firstly, it is determined whether or not the particle is interacting with the body, using a simple distance calculation and comparison. If the body resides inside of the body (this is possible due to integration (advection) error) it must be pushed out to the edge of the body, and adding a velocity due to the particle's vorticity at the point where it would otherwise make contact with the body. After this velocity has been updated, it is simple to re-update both the vorton's position and its vorticity due to the newly calculated velocity at the contact point. One thing to note is that, after completing the advection stage of the simulation, it is more efficient to then store the interpolated cell velocity as a velocity element of the vorton itself so that it does not require the velocity to be re-interpolated when calculating the velocity caused by the interaction with the body.

### 6.4.6 Fire Performance Comparison

The comparison between the velocity-vortex method and the non-interacting particle method is fairly straightforward. Data for the comparison is shown in the graph in Figure 6.5. As shown, the non-interacting particle model is much faster than the velocity-vortex model. Because of the simplicity of the non-interacting kernel, there is very little error in the results (as compared to the velocity-vortex method).

One possible option to consider is removing part of the randomization element of the velocity-vortex method. The simplicity of the non-interacting algorithm requires it to have the randomization element, but the velocity-vortex methods interaction over time could produce semi-random behaviour using set starting positions. To be clearer, particle attributes would be randomized at initialization but instead stored as constant values which are reset when the par-

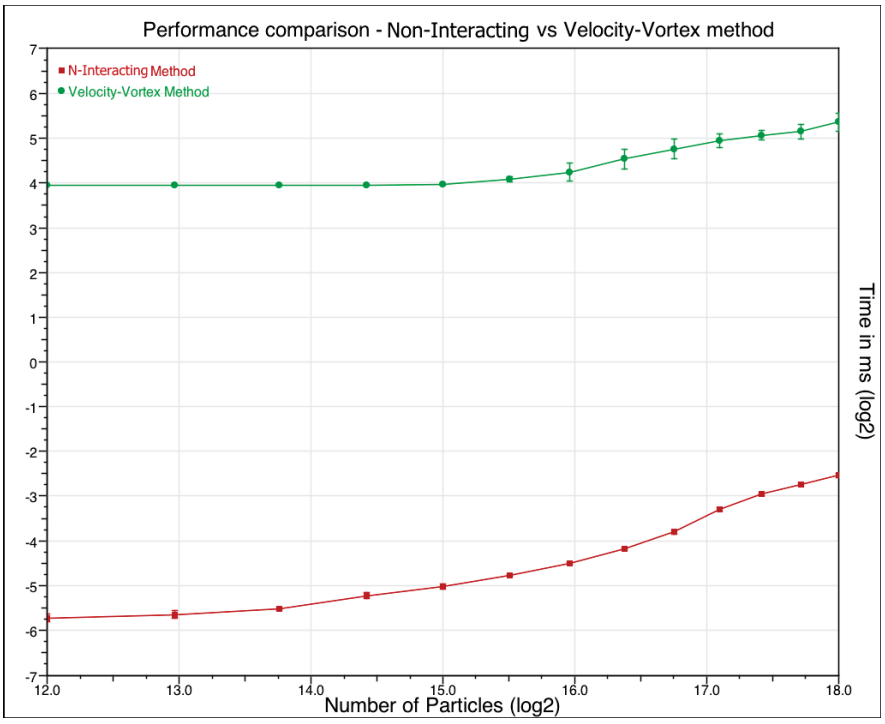


*Figure 6.5: Performance comparison of particle methods for simulating a flame. Red points are the results for the non-interacting method, while green are the velocity-vortex results. Times are averaged over 50 runs, with 1000 iterations for each run, with a 1000 iteration grace period after initialization to account for any instability due to initial randomization.*

ticle is destroyed. However, because the age and decay of the particle is randomized, when it is recreated the positions of the particles around it will be different and subsequently the induced vorticity will also be different, causing the particle to move along a different path than before. This could save some computation time on the velocity-vortex method while making little to no difference in the visual aspect of the simulation.

Again, the visual aspects of the fire simulation will be discussed later in Section 7.1. Figure 6.6 shows a sample image of the velocity-vortex fire system as a reference, showing the general shape created by the method. The rendering method uses a point-based sphere approach simply to show the positions of the particles, and particles are coloured randomly with a red colour ramp. A more in-depth point-based method for rendering the fire is discussed later in the chapter.

## 6.5 Water Implementation

The water implementation features two main methods. The modified spring-mass particle system simulates water in a relatively simple way by calculating soft-body collision forces between particles on collision using spring-mass methods, while ignoring the spring extension forces by allowing each particle to move freely away. The larger the number of particles there are, the better the illusion of fluid-like behaviour. The second method, the Smoothed Particle

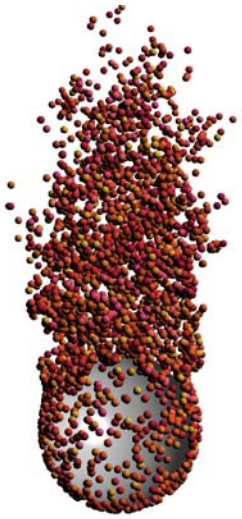

*Figure 6.6: Rendering of the fire using the simple point-based sphere approach to show the general positions of the particles and the collision sphere. Colours are randomized on a red-orange colour ramp.*

Hydrodynamics (SPH) method, uses far more accurate fluid-dynamical equations to govern the behaviour of the fluid, accounting for viscosity, pressure and density of particles within. These two methods will be compared later in the chapter for the advantages and disadvantages of each when simulating and rendering in real-time.

### 6.5.1 Spring-Mass System Implementation

The spring-mass particle model is most often used in cloth simulation [108] [32] [7], but modifying the method slightly can give a number of different effects. In particular, spring-mass equations can be used to model a simple fluid simulation by adding certain restrictions on the forces being applied.

As particles move towards each other and collide, a repulsion force is applied using soft-body collision dynamics. If the distance between two particles is smaller than the sum of the two particles' radii, they must be colliding at this particular timestep. An elastic spring force can be applied by assuming the length of the spring at rest is equal to the sum of the two particles' radii. Thus, as a collision occurs between particles, the spring is essentially compressed the distance of the the rest distance minus the current distance between the two particles at collision. It should be clarified that "compression" technically does not happen - rather, the discrete timestep of the integration method has caused both particles to partially occupy each other's space. The force which subsequently separates these two particles, however, is calculated based on the standard spring-force calculations given by Hooke's Law [113] and includes shear, drag and damping forces, as described in Equation 4.6.

The spring-mass method for simulating fluids is not as accurate as other methods because there are no fluid dynamics equations used. This means that physical behaviour such as fluid

viscous forces are not considered. Nevertheless, the method can still provide a surprisingly good effect especially when the number of particles in the simulation is relatively large. The effect produced by this method would be similar to the behaviour of plastic balls in a child's ball pen, or a bucket of marbles.

Much of the code involved is calculating the spring-mass forces is similar to that in Listing 6.4. Pseudo-code is highly preferable in this case, where the underlying basics of the code remain the same for all of the simulations, and better descriptive context is required for the subtle, but important, force calculations. The pseudo-code for the application of the spatial grid to spring-mass model has already been described in Algorithm 3 in Chapter 4, although it does not elaborate on the force calculation, instead focusing on a more general overview. Algorithm 11 shows a more precise description of the method.

There are a couple of things to note about this algorithm. Firstly, it is assumed that the radius and mass of both particles, which are colliding in any one instance, are identical. This is an acceptable assumption for the water simulation. Secondly, it is assumed both particles have the same material properties and therefore use the same force constants. This is fine for simulating a single fluid, but simulating the interactions between multiple particle types (for example mixing two fluids) must take into account all sets of different material properties.

As shown in Algorithm 11, the spatial grid is used to simulate the spring-mass system in real-time (similar to the plasma simulation) by dividing the simulation space up into manageable chunks. Unlike the plasma system however, the size of the spatial grid cell does not need to accommodate multiple particles. The optimum spatial grid cell size should be twice the actual particle radius. This means that at any one time, there should technically only be one particle allowed within any one cell. In practice, this does not hold one hundred percent of the time as again, due to the discrete timestep of the integration method, particles may be able to move far enough into the next cell. In any case, the number of potential force calculations is greatly reduced and times where more than around 26 calculations per timestep (one per neighbouring spatial grid cell) are few and far between.

When calculating the spring and shear forces, notice that force modifications only occur when the distance $d$ between the two particles is less than the rest distance $2R$. This means that once the particles have collided and rebounded, they are free to move away without any spring-

| Parameter | Value |
|---|---|
| Particle Number | 512,000 |
| Timestep | 0.005 |
| Spring Constant | 0.4 |
| Shear Constant | 0.2 |
| Damping | 0.95 |
| Gravity | 9.8 |
| Particle Mass | 1.0 |

*Table 6.1: Table with simulation parameters of the spring-mass model for water simulation*

**Algorithm 11** In-depth spring-mass particle model force calculation algorithm

Given array P of *n* particle positions
Given array V of *n* particle velocities
Given arrays F and L of sorted particle indices (First and Last)
Given simulation constants K - spring, shear and damping
Given particle radius R and mass M
Acceleration A (force) = 0
//Spatial grid arrays have already been sorted
//Current cell position is a function of particle position P
**for** each particle in PARALLEL, with thread index *i* **do**
  Cell C = func(P[*i*])
  //Find neighbours. Neighbouring cells are -1 to +1 in all directions
  **for all** -1 $\leq$ x $\leq$ +1, -1 $\leq$ y $\leq$ +1, -1 $\leq$ z $\leq$ +1 **do**
    Neighbour N = C + Vec(x, y, z)
    //Use neighbouring cell indices to obtain list of neighbouring particles in F and L
    *cell_index* = cellfunc(N)
    *f* = F[*cell_index*]
    *l* = L[*cell_index*]
    //Loop through all neighbouring particles
    **for** *p* = *f*; *p* $\leq$ *l*; *p*++ **do**
      Relative position $P_{rel}$ = P[*i*] - P[*p*]
      Let *d* be the length of the vector $P_{rel}$
      //Collision occurs when d is less than the *collision-* or *rest distance*
      //If two particles are equal size, collide when *at most* twice the radius R
      **if** d $<$ 2R **then**
        Relative velocity $V_{rel}$ = V[*p*] - V[*i*]
        //Rebound direction (normal) is the inverse of the normalized $P_{rel}$
        norm = -$P_{rel}$ / *d*
        Tangental velocity $V_{tan}$ = $V_{rel}$ - dot($Vrel$, norm) * norm
        //Acceleration - spring along norm, shear along $V_{tan}$, damping along $V_{rel}$
        A[*i*] += -$K_{spring}$ * (2R - *d*) * norm
        A[*i*] += $K_{shear}$ * $V_{tan}$
        A[*i*] += $K_{damp}$ * $V_{rel}$
      **else**
        No collision, no force is applied
      **end if**
    **end for**
  **end for**
  //F = ma
  A[*i*] *= M
  Add external force E where applicable
**end for**
All forces have been calculated. All new particle data is passed to integration method

extensive force applied. After the spring-mass force calculation is complete, external forces are applied to produce the final force calculation and then the integration step is performed. Table 6.1 shows the simulation parameters used for a spring-mass model to produce a standard water simulation.

### 6.5.2 Smoothed Particle Hydrodynamics Implementation

The SPH fluid simulation implementation is different to the spring-mass model in two main ways:

- The usage of the fluid dynamics equations to calculate extra acting forces on the particles
- The calculation of pressure and viscosity requires a mass-density calculation on *all* particles to be performed beforehand, as the total density must be summed for all neighbouring spatial grid cells before the pressure or viscosity on the actual particle can be calculated.

The first point is relatively straightforward - rather than a single spring-force calculation, the sum of all forces acting on the particle is comprised of both pressure and viscosity forces in the SPH simulation. The second point however, means that the SPH simulation requires the spatial grid to be passed through *twice*. The first pass calculates the density for each particle, and the second pass calculates the pressure and viscosity forces. Essentially, this means that the SPH method will effectively run slower, but this is not always the rule as will be explained shortly. Apart from these two aspects, the SPH implementation of a water simulation remains largely unchanged. Similar to the spring-mass model, Algorithm 12 shows a more in-depth look at the calculation of the forces in the SPH simulation.

There are several things to note in Algorithm 12. As with the spring-mass model, the mass of all particles is assumed to be the same as well as the fluid parameters. The algorithm performs two passes through the spatial grid, firstly calculating the density and pressure values for each particle in the fluid, and then doing a second pass through the grid and using those values to calculate the pressure and viscosity forces acting on the particles.

| | Pseudo Code | Value |
|---|---|---|
| Particle Number | n | 524,288 |
| Timestep | t | 0.0005 |
| Particle Mass | M | 0.00005 |
| Rest Density | $K_{restD}$ | 1000.0 |
| Rest Pressure | $K_{restPr}$ | 0.0 |
| Gas Stiffness | $K_{gasSt}$ | 1.5 |
| Smoothing Kernel Length | $K_{len}$ | 0.00641 |

*Table 6.2: Table with simulation parameters of the SPH model for water simulation, with pseudo-code symbols.*

**Algorithm 12** In-depth SPH model force calculation algorithm

Given array P of *n* particle positions
Given array V of *n* particle velocities
Given array D of particle densities
Given array Pr of particle pressures
Given arrays F and L of sorted particle indices (First and Last)
Given simulation constants K - kernel length, length squared, rest density and pressure, gas stiffness, kernel coefficients
Given particle mass M
Accelerations (forces) $A_p$ (pressure force), $A_v$ (viscosity force), $A_{total}$ = 0
**for** each particle in PARALLEL, with thread index *i* **do**
  Cell C = cellfunc(P[$i$])
  //Find neighbours
  **for all** -1 $\leq$ x $\leq$ +1, -1 $\leq$ y $\leq$ +1, -1 $\leq$ z $\leq$ +1 **do**
    Neighbour N = C + Vec(x, y, z)
    *cell_index* =cellfunc(N)
    *f* = F[*cell_index*]; *l* = L[*cell_index*]
    //Loop through all neighbouring particles
    **for** *p* = *f*; *p* $\leq$ *l*; *p*++ **do**
      D[$i$] = **calc_density**(P[$i$], P[$p$], $K_{lenSq}$, $K_{sm}$, M)
      //For SPH, use the smoothing kernel constants to calculate density
    **end for**
  **end for**
  //Final density and pressure is calculated using SPH equations
  D[$i$] = max(1.0f, M * $K_{poly6co}$ * D[$i$])
  Pr[$i$] = $K_{restPr}$ + $K_{gasSt}$ *( D[$i$] - $K_{restD}$ )
**end for**
//Now use pressure and density to calculate forces
**for** each particle in PARALLEL, with thread index *i* **do**
  Cell C = cellfunc(P[$i$])
  //Find neighbours
  **for all** -1 $\leq$ x $\leq$ +1, -1 $\leq$ y $\leq$ +1, -1 $\leq$ z $\leq$ +1 **do**
    Neighbour N = C + Vec(x, y, z)
    *cell_index* = cellfunc(N)
    *f* = F[*cell_index*]; *l* = L[*cell_index*]
    //Calculate pressure and viscosity forces using SPH equations
    **for** *p* = *f*; *p* $\leq$ *l*; *p*++ **do**
      $A_p$ = **calc_pressure_force**(P[$i$], P[$p$], D[$i$], D[$p$], Pr[$i$], Pr[$p$], $K_{len}$, $K_{sm}$, M)
      $A_v$ = **calc_viscosity_force**(P[$i$], P[$p$], D[$i$], D[$p$], V[$i$], V[$p$], $K_{len}$, $K_{sm}$, M)
    **end for**
  **end for**
  //Finalize SPH forces
  $A_{total}$[$i$] = $A_p$ * $K_{kernelPr}$ + $A_v$ * $K_{kernelV}$ * M
  Add external force E where applicable
**end for**
All forces have been calculated. All new particle data is passed to integration method

```
//Calculate the density of particle A from B
__device__
float calc_density(float3 posA, float3 posB,
                   float k_length_sq, float smoothing_coeff,
                   float particle_mass){

    //Using the poly6 smoothing kernel
    float3 rel_pos = posA - posB;
    float dist = length(rel_pos);
    float dist_sq = dist * dist;

    //((h^2 - r^2)^3
    float d = k_length_sq - dist_sq;
    d *= d * d;

    //coeff: 315/64*pi* h^9
    return particle_mass * smoothing_coeff * d;
}
```

*Listing 6.7: Calculating the particle mass-density. See previous equations in Section 4.3 for a more detailed explanation*

Many constant simulation parameters are used in this method, many of which have been pre-calculated during the particle system initialization and put into constant memory. These parameters are explained in more detail in Table 6.2. The interesting parts of the algorithm are highlighted in bold. Three device kernels, `calc_density`, `calc_pressure_force` and `calc_viscosity_force` contain the bulk of the SPH functionality. Each of these device kernels are shown in detail in code listings 6.7, 6.8 and 6.9 respectively.

Table 6.2 shows the simulation parameters used for a SPH model to produce a standard water simulation. For reference, it contains the equivalent symbols used in the pseudo-code in Algorithm 12. These parameters can be easily changed to get vastly different behaviour from the fluid. Different smoothing kernel coefficients are calculated directly based on the smoothing kernel length $K_{len}$.

Listing 6.7 shows the device kernel used to calculate the density of each particle in the fluid. The square of the kernel smoothing length `k_length_sq` and the smoothing coefficient are constant values which have been pre-calculated and stored in constant memory for fast access. The `posA` and `posB` are the positions of the current particle and neighbouring particle respectively. Once the density has been calculated, the pressure of the particular particle can be calculated using Equation 4.11 (see Chapter 4).

The calculation of the force due to pressure is shown in Listing 6.8. It is using a conservation of momentum equation to make sure that the force is symmetrical - both particles will apply the same pressure force on each other. Taking the positions of the two particles as well as the pressure and density of these particles, it uses the first differential of the spiky smoothing kernel to calculate the final pressure force.

The viscosity component of the force is also calculated as shown in Listing 6.9. Taking the position and velocity of the two particles it uses the laplacian of the viscosity smoothing kernel to calculate the viscosity force.

```
//Calculate the pressure force of particle B on A
__device__
float3 calc_pressure_force(float3 posA, float3 posB,
                           float pressureA, float pressureB,
                           float densityA, float densityB,
                           float k_length, float smoothing_coeff,
                           float particle_mass){

    //Using the first differential spiky smoothing kernel
    float3 rel_pos = posA - posB;
    float dist = length(rel_pos);

    float p = (pressureA / (densityA * densityA)) +
              (pressureB / (densityB * densityB));

    //((h - r)^2
    float s = k_length - dist;
    s *= s;

    //coeff: -45/pi* h^6
    return p * smoothing_coeff * (rel_pos / dist) * s;
}
```

*Listing 6.8: Calculating the particle pressure force. See previous equations in Section 4.3 for a more detailed explanation. Smoothing kernel length, coefficient and mass are all pre-calculated and stored in constant memory for fast access.*

```
//Calculate the viscosity force of particle A from B
__device__
float calc_viscosity_force(float3 posA, float3 posB,
                           float3 velA, float3 velB,
                           float densityA, float densityB,
                           float k_length, float smoothing_coeff,
                           float particle_mass){

    //Using the laplacian viscosity smoothing kernel
    float3 rel_pos = posA - posB;
    float dist = length(rel_pos);

    float3 v = (velB - velA) / (densityA * densityB);

    //h - r
    float s = k_length - dist;

    //coeff: 45/pi* h^6
    return v * smoothing_coeff * s;
}
```

*Listing 6.9: Calculating the particle viscosity force. Viscosity force is dependent upon the velocity of the particle - a faster velocity means a greater viscosity force, because fluid dynamics seeks to keep particles from escaping the fluid. Smoothing kernel length, coefficient and particle mass are all pre-calculated and stored in constant memory for fast access.*

Afterwards the total force is added up and multiplied by the assumed constant particle mass, as well as any external forces such as gravity and boundary forces. Once this final force has been calculated it is used in the integration method in calculating the movement of the particles. This completes the implementation of the SPH system for water. All that is left to do is render the system.

### 6.5.3 SPH vs Spring-Mass Comparison

The comparison between the two methods for simulating water is done based on the following criteria:

- Kernel execution times per frame for varying sizes of the particle system.
- Qualitative comparison - visual aspect

A comparison of the execution times of both methods shows different strengths and weaknesses depending on the number of particles being used in the particle system. The comparison was performed using the leapfrog integration method and using varying numbers of particles as the system size; from $2^{14}$ (16,384) at the lowest to $2^{19}$ (524,288) at the highest.

As shown in Figure 6.7, the spring-mass model outperforms the SPH method up until around $2^{17}$ (131,072) particles. This is inherently due to the fact that the SPH method requires the second run through of the spatial grid when calculating the pressure and viscosity forces, after already calculating the density and pressure values for each particle. When using larger system sizes than this, however, the SPH method starts to outperform the spring-mass method. The reason for this is that even with more and more particles in the system, the SPH method continues propagate pressure forces consistently through the entire fluid, keeping particles apart, as well as using the viscosity forces to keep surface particles from escaping the fluid. The spring-mass model does not do this, so it consequently suffers performance issues when larger system sizes are used. With more particles in the system, it becomes easier for particles to push into neighbouring grid cells, causing more particle-particle interactions to occur per simulation iteration. The reason why it is easier in the spring-mass system is because there are no pressure forces from particles in the neighbouring area pushing against other incoming particles - reactions only occur in *direct* contact with others. This effect is compounded as the level of error introduced by the integration method (even using a good one) is increased as the system size is increased. Additionally, the lack of viscous forces can lead to relative instability in the fluid, as particles with extraordinary velocities can break out of the fluid and impact other areas, which would have been counteracted if a viscous force had kept the breakaway particle securely in the fluid.

The visual results are displayed in Figure 6.8 using a simple sphere shader. It shows the visual differences in the two different methods. The SPH gives a better effect of water simulation. The wave-like effects are more pronounced for the SPH version of the simulation, although the effect still works ok for the spring-mass system. Wave-like fluid shapes are able to be made by the SPH method because of the viscosity forces keeping the fluid together. In

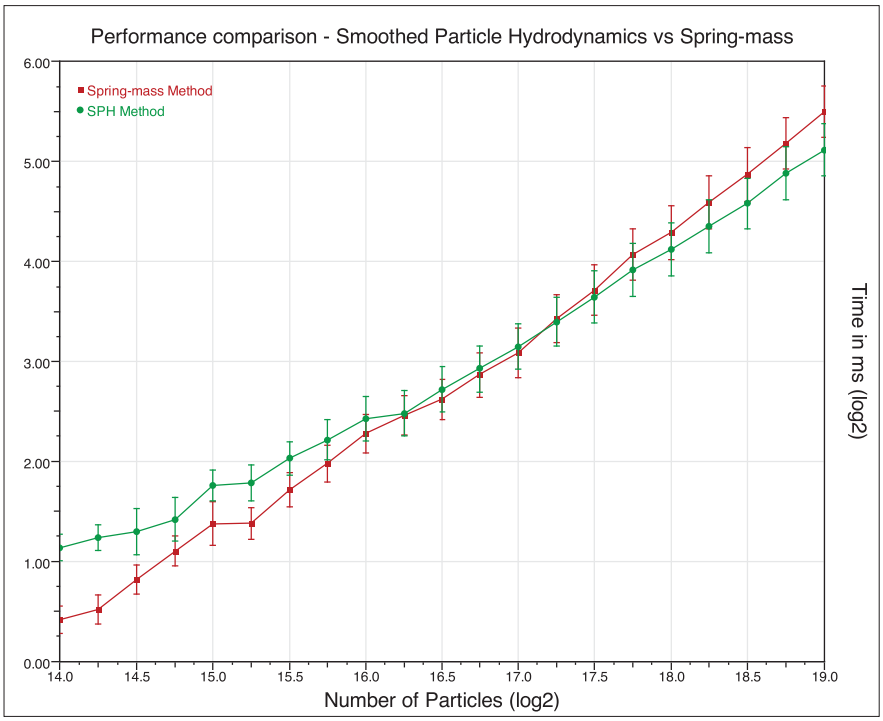


*Figure 6.7: Comparison of the execution times of both the SPH and spring-mass methods for simulating fluid behaviour. The scenario for this simulation is that fluid particles are all initialized randomly in the center of the environment and allowed to fall to the ground and spread out until the fluid has reached a somewhat settled state. The execution times are sampled over 2000 iterations and is averaged over 50 separate runs. The error in the graph occurs due to the dependent paths of the particles with randomized initial positions.*

contrast, the spring-mass method only applies gravity forces once particles are freed from the fluid, resulting in a flatter wave. The spring-mass method would be much more suitable in a situation where less complex fluid motion is required - for example, in simulating tides or waves coming into a beach would work well for this method. The SPH method can be used for much more turbulent fluid motion. The following Figure 6.9 contains a series of screenshots using both the spring-mass system configuration and the SPH configuration. The screenshots are set up as a time lapse series and is intended to explain the visual differences in the behaviour of the two different methods. Additionally, these screenshots show the system rendered using a surface generation approach, which will be discussed in the next chapter.

## 6.6 Summary

This chapter covers the implementation of each of the different particle systems for fire and water. Each of these systems are explained in terms of computational implementation using CUDA. For the fire simulation, a non-interacting particle model is implemented, as well as a velocity-vortex particle model. In comparing these results for performance, the non-interacting model is found to be faster, however it will be shown later in Chapter 7 that the visual results

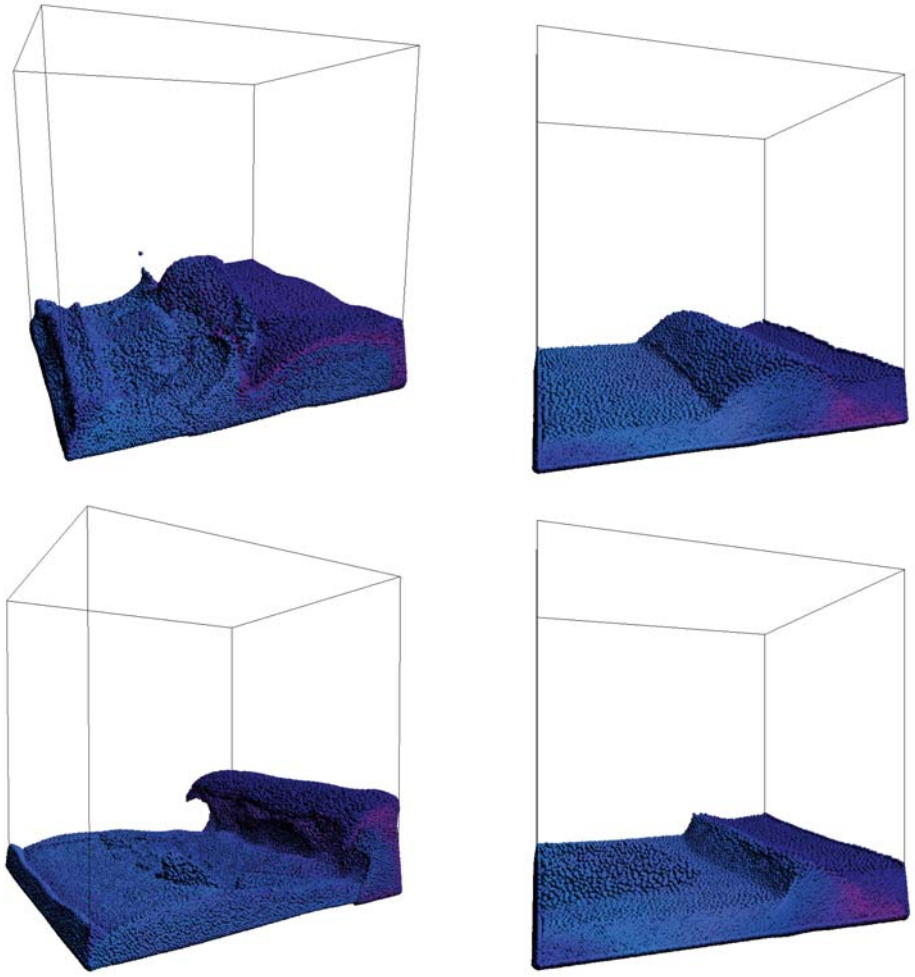

*Figure 6.8: Screenshots of the water simulation showing both methods - the left-hand images are produced by the SPH particle system, while the right are produced by the spring-mass system. The size of the particle system in these screenshots is $2^{19}$ particles. The scenario used in this case is that particles are dropped in a concentrated area along the left-hand side of the simulation space (relative to the camera), allowing particles to wash back and forth in a wave-like motion.*

are actually in favour of the velocity-vortex model. For the water simulation, the spring-mass model was implemented, as well as the SPH model. It can be shown that the SPH model performs better for larger system sizes, and better represents the fluid-like behaviour of water as shown in Figure 6.8. In the next chapter, each of these systems will be rendered using OpenGL and CUDA interoperability. Simple point-based approaches will be described and variations on this method will be implemented for the fire systems. The water systems are rendered slightly differently - as a *free surface*. The visual results of generating this surface, as well as the challenges involved, will be explored.

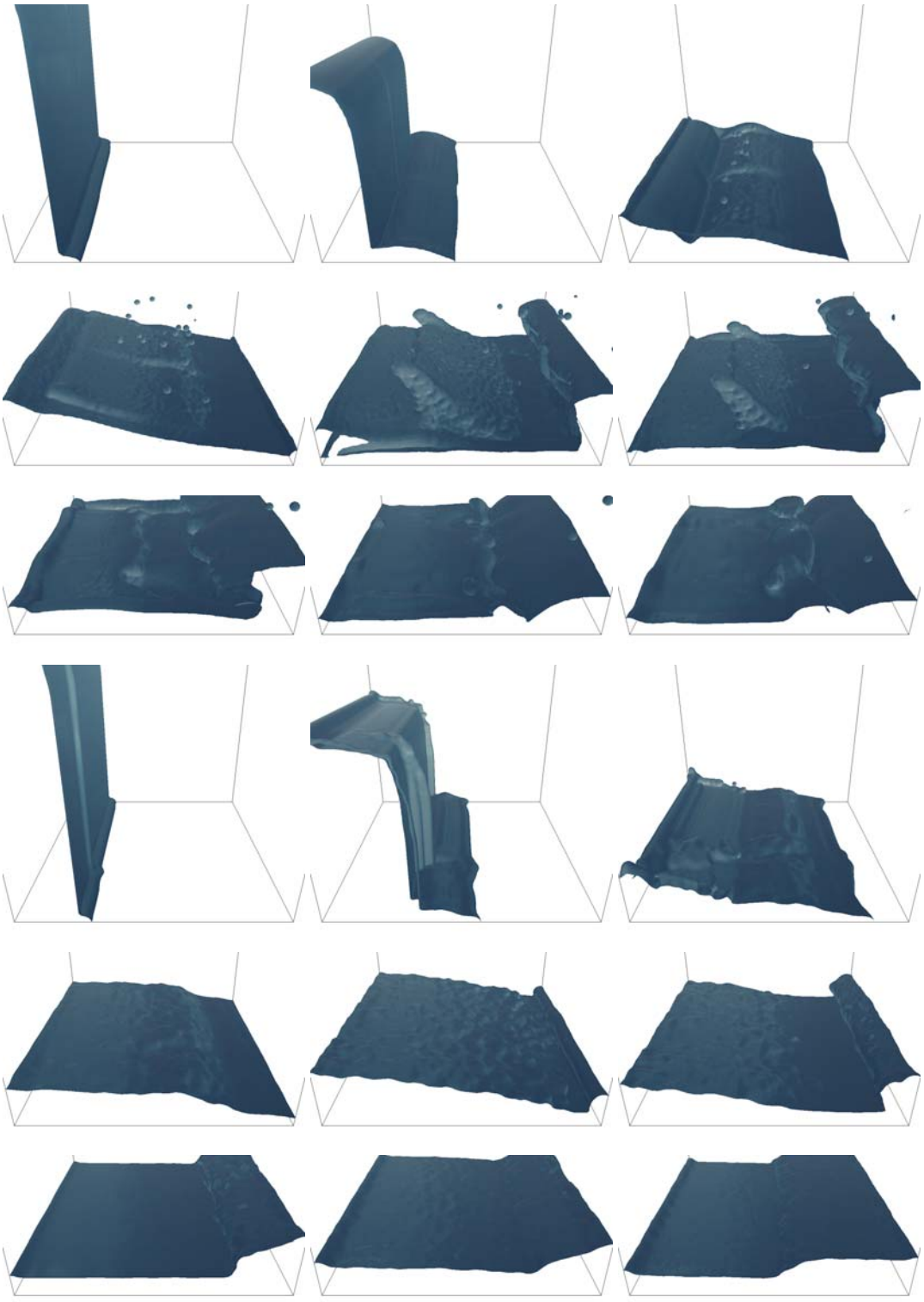

*Figure 6.9: Time lapse showing the differences between the two models used for the water particle system, rendered using a surface generation and fresnel reflection approach. A screenshot of the simulation is taken every 100 iterations. The first set of 9 screenshots belongs to the SPH model, while the second set of 9 belongs to the spring-mass model. These time lapses clearly show the behavioural differences of the two models.*

# CHAPTER 7

# RENDERING

It is explained in Chapter 6 how the different particle system methods are implemented, parallelizing the model's algorithms and producing good quality behaviours from the different approaches. For fire systems, the non-interacting method performs faster than the velocity-vortex method, but how do these two methods compare visually? Since the velocity-vortex method stores more available data for rendering purposes, this method is more flexible when it comes to rendering. For water systems, the SPH model provides better results than the spring-mass model, however the spring-mass model is still able to simulate the water reasonably effectively.

The rendering methods of the different particle systems vary greatly depending on which of the phenomena is being simulated. Unlike water, fire effects tend not to have a specifically defined free surface. Although it is technically possible to render the fire system with this method (all that is required for the surface to be generated are the positions of the particles, which are observed by all particle systems featured in this thesis), the surface does not visually represent a realistic fire or flame very well. For rendering fire then, a sprite-based approach is used to produce a "fuzzy" effect. Surface rendering is only applied to the water system, however the behaviour of the water can still be observed using simpler point-based methods and the comparison between these can easily be seen - for example, in the comparison screenshots in Figure 6.8.

In this chapter, three different rendering methods will be explored. A general overview of rendering in OpenGL will be provided in Section 7.1. In Section 7.2 a simple point-based method will be described, as well as variations of this method for different effects. Section 7.3 then compares the two fire system implementations using these point-based methods. In Section 7.4, a volumetric rendering approach will be explored, while in Section 7.5 the method for generating surfaces will be described in detail. The generation of surfaces for water simulation is particularly important, as it explores the challenges of merging both a complex computational method and a complex rendering method. Visual results for the surface generation approach to the water particle system will be provided in Section 7.7.

## 7.1 Rendering Introduction

The rendering of simulations is unnecessarily limited when data is needed to be copied from the device to the host every time a frame is needed to be rendered. Rendering performance can be improved by having the rendering data remain on the graphics card at all times without needing to take precious time in copying over the device data. When using the OpenGL rendering API, this is what is known as *OpenGL-CUDA interoperability*.

There are coding methods in OpenGL which allow data to be read directly on the GPU. Data are stored in *object buffers* and can be accessed by mapping or preparing the buffer for use with OpenGL. Buffers can only be used one at a time. Data being operated on by CUDA kernels cannot be used by the OpenGL API at the same time, so CUDA must release control of the buffer so that OpenGL can use it. Similarly, OpenGL must also release control of the buffer before CUDA can resume the parallel operations on the data.

Vertex Buffer Objects (VBOs) are used to store arrays of data on the graphics card and then access them directly. A general example would be to store the positions of the particles as an array of floats into a position VBO. If using a simple point-based rendering approach (See Section 7.2), OpenGL can then access the data by binding the buffer containing the position values to the array buffer used to render the graphics elements. Thus OpenGL does not need to copy the data off the GPU onto the host before it renders it. A particle's position does not always need to be stored in a VBO; for example, if a surface is to be drawn *over* the particle, the VBO would contain the positions of the vertices making up the surface mesh, not the particle's position itself. See Chapter 7 for a more in-depth explanation. VBOs can also store additional data information relating to each vertex. It is common to use an additional buffer object to store colour values when using point-based rendering methods, while when rendering three-dimensional meshes, the surface normal at each vertex is often provided to be used when calculating lighting or reflections, and texture coordinates are needed when a texture is to be mapped to the surface.

CUDA can access VBO data by registering the buffer to a `cudaGraphicsResource` structure. See Listing 7.1 for an example. Once the buffer has been *mapped* to CUDA using a mapping function, only CUDA can access the data, using it in the CUDA kernels such as the particle integration methods and the spatial-grid based sorting. Before the system is to be rendered, CUDA must un-map the buffer so that it can be used by OpenGL again. The system is rendered, and the process starts again.

In graphics a *shader* is a program which is run on the GPU to perform some graphics function. Normally this means completing the *shader* stage of the graphics rendering pipeline. Shaders are written in a *shading language* - the most common shading languages are *GLSL* (*OpenGL Shading Language*) and DirectX's *HLSL* (*High Level Shading Language*). Both languages are very similar and are designed to create shader programs resembling traditional C code. GLSL is used in this thesis.

There are two main types of shaders - *vertex* and *fragment* shaders. Vertex shaders handle the processing of any vertex data, such as particle position data in VBOs. Fragment shaders

```
//Create the VBO in the initial stages of the program
uint createVBO(uint size, struct cudaGraphicsResource **resource){

    GLuint vbo;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    cudaGraphicsGLRegisterBuffer(resource, vbo, cudaGraphicsMapFlagsNone);
    return vbo;
}
```

*Listing 7.1: Initializing VBOs for use with CUDA/OpenGL*

handle data used after the rasterization process for determining the colour of pixels. When using a shader model, vertex and fragment shaders are required. Other secondary shaders such as the *geometry shader* are optional.

## 7.2 Point-Based Rendering Method

Point-based rendering methods are some of the simplest rendering methods that can be used with particle systems. The basic idea is to use the particles position as the center point to draw a two-dimensional texture or *sprite* representing the particle. The texture is rotated such that it always faces the camera as the camera moves around the system, giving the impression that the particle is three-dimensional. This method is often referred to as the *billboarding method* and is a popular method in computer graphics for improving performance as sprites can be processed a lot faster than actual three-dimensional models. Sprites are particularly effective for particle simulations as particles are very rarely rendered individually, but rather larges groups comprising of multiple combined or overlapping sprites, giving the impression of the system having some sort of volume.

Listing 7.2 demonstrates a basic point-based vertex shader. Along with the camera's position, the `point_size` variable is used to increase the size of the sprite the closer the camera gets to the particle. The position of the particle is outputted to `gl_Position` while the size of the sprite is outputted to `gl_PointSize`. The colour of the particle is passed out to be used by the fragment shader. All OpenGL matrix transformations are also loaded in manually as uniform variables but were omitted from the listing for the sake of brevity.

If particles are intended to bounce off each other (such as the in spring-mass model), the point size setting in the spherical vertex shader must be set such that no two particles overlap during a collision. This method does not apply in the case of the SPH model, however. This is because in this situation, the particles themselves are treated as interpolation points over which the entire fluid is sampled, instead of individual interactive objects. This results, for example, in many particle sprites overlapping each other in highly compressed areas. This is not ideal, but for the purpose of understanding the positions of these points, this rendering method is sufficient.

```
//Matrix transformations also passed as uniform variables
uniform float point_size;
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec4 colour;
out vec4 vs_colour;
void main(){
    vec4 position = vec4(vertex, 1.0);
    gl_Position = ModelViewProjectionMatrix * position;

    //Modify the point size depending on the distance to the camera
    vec3 eye_pos = vec3(ModelViewMatrix * position);
    gl_PointSize = point_size / length(eye_pos);
    vs_colour = colour;
}
```

*Listing 7.2: Point-based vertex shader. The vertex and colour attributes are provided by the respective buffer objects. Uniform variables are constant values passed into the shader by the host. The points are scaled as the camera moves closer to the particle, using the length of the eye vector to change the size of the point sprite, with the base size of the sprite being dependent on the particle radius.*

```
//Fragment shader for point sprites
uniform sampler2D sprite_tex;
in vec4 vs_colour;
layout (location = 0) out vec4 out_colour;
void main(){
    vec4 tex_colour = texture(sprite_tex, gl_PointCoord);
    out_colour = vs_colour * tex_colour;
}
```

*Listing 7.3: Point-based rendering fragment shader, used for rendering the fire system. Using alpha textures and blending allows for the particles to appear fuzzy in places where there are few particles, and intense in places where particles are concentrated.*

```
//Sphere sprite fragment shader
uniform vec3 light_dir;
in vec4 vs_colour;
layout (location = 0) out vec4 out_colour;
void main(){
     //Find vector from the fragment to the center of the point sprite
     vec3 N; N.xy = gl_PointCoord * vec2(2.0, -2.0) + vec2(-1.0, 1.0);
     //Find this distance and discard any fragments outside circle
     float mag = dot(N.xy, N.xy);
     if (mag > 1.0) discard;
      N.z = sqrt(1.0-mag);
     //Add lighting from input direction (diffuse light)
     float light = max(0.0, dot(light_dir, N));
     out_colour = vs_colour * light;
}
```

*Listing 7.4: Point-based sphere fragment shader. Draws a sphere at the vertex point given by Listing 7.2. OpenGL point sprites must be enabled for this to work, and the size of the spheres is changed in the vertex shader by modifying the point-size uniform variable. The light direction is applied here also as a uniform variable which may be changed over time or remain static throughout the simulation. This shader is used for the demonstration screenshots, not for the fire simulation itself.*

*Figure 7.1: Example texture used for alpha values in point-based rendering*

The fragment shader in Listing 7.3 takes the colour from the colour buffer object (passed to it from the vertex shader) as well as a two-dimensional texture containing the alpha values for the colour element. This texture is what determines the shape that the individual particle will look like. The texture might look something like Figure 7.1. The combination of the input colour from the colour buffer values and the alpha value from the texture will produce the final fragment colour.

Lighting effects can also be applied in the fragment shader stage. Listing 7.4 shows an example of the OpenGL shader used to render a spherical sprite for each particle, modified slightly from the sphere shaders used in Green's particles demo [41]. Rather than load in a texture, the sprite is rendered by drawing a coloured circle and applying some arbitrary light to part of it to give the illusion that it is three dimensional.

Shading pixels this way is a much more efficient method to using three-dimensional sphere models to render each particle, as there is no need to draw large numbers of polygons to render the spheres. Variations of this shader are used in the majority of the screenshots in this thesis where individual particles are displayed, as this shader makes it very easy to distinguish these individual particles even when system sizes are relatively large.

## 7.3 Fire Visual Comparison

The two fire implementations are compared using the point-based shaders as described above. The comparisons are shown in Figure 7.2. The rendering method employs point sprites based on the Figure 7.1. However, the two approaches are slightly different between the two methods.

The non-interacting system requires randomization to create the desired effect - the correct colours are chosen by biasing certain colours towards certain emission areas - red colours are given to particles on the edges of the system, yellow colours are given to particles closer to the center. Additionally, the alpha colour values for each particle are changed as the particle ages - the particle fades away more and more as it rises until it is destroyed and recreated. Using

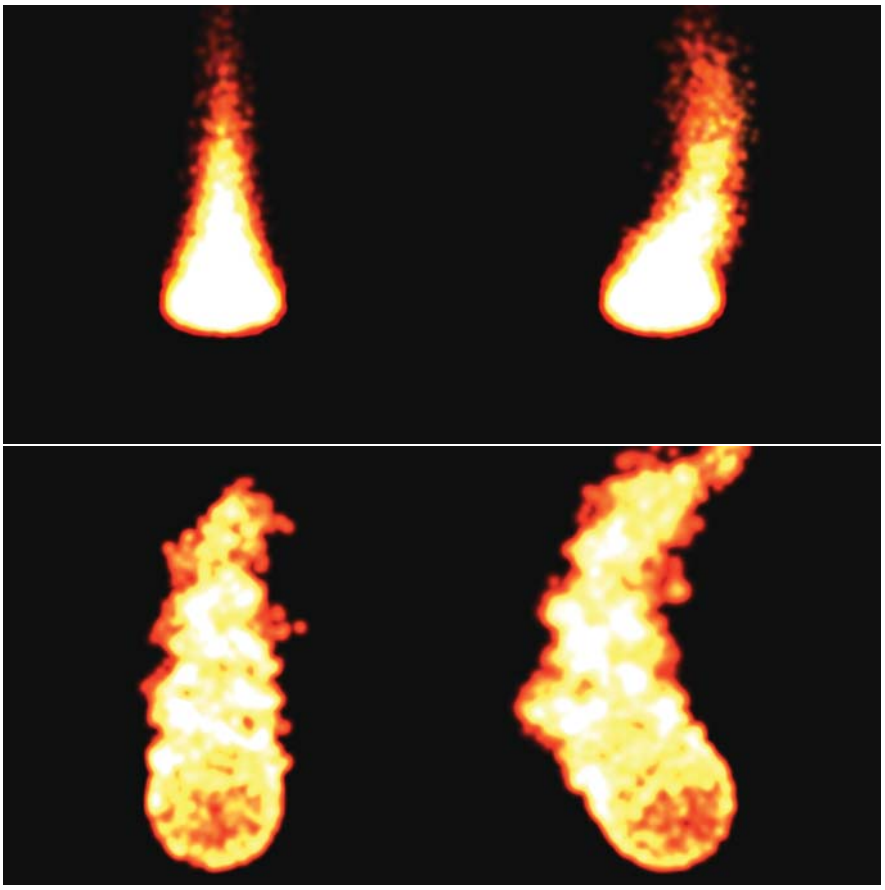

*Figure 7.2: Visual comparison of the different fire particle systems rendered using a point-based texture shading approach. Top left: Non-interacting system, calm. Top right: Non-interacting system, with wind. Bottom left: Velocity-vortex system, calm. Bottom right: Velocity-vortex system, with wind.*

these values creates the desired effect but in terms of flexibility the method is fairly restrictive - if a different emission area is defined for instance, a completely new set of rules determining the colours and alpha values may need to be created.

In contrast, the velocity-vortex method uses the additional fluid density attributes of the particles in the fluid to colour the sprites. Particles in higher density areas in the fluid appear brighter than those in lower density areas. In this way, edges of the flame are automatically more red-coloured, as the density here is relatively low, while areas in the center of the flame are bright due to the fluid being high in density. Note that the area around the rigid-body emitter appears to have relatively low density (red) as there are no particles contained inside the rigid-body and the body itself is not rendered in this instance. It can be shown from the comparison of these screenshots that the velocity-vortex method gives a better visual representation of the flame effect than the non-interacting method.

Additional rendering methods for fire are also available - most notably, the *black-body radiation* method for rendering heated gases. This method involves using Planck's Law [104] to correctly calculate the radiant intensity of the fluid for different frequencies of light. This

method was used in the volume rendering approach used by Horvath and Geiger [56]. In order to use black-body radiation rendering for this fire, the velocity-vortex method would need to be extended to implement some combustion mechanics, so that the correct temperatures can be calculated and used in the black-body radiation function, or at least provide temperature values which sufficiently approximate the values needed.

## 7.4 Volumetric Rendering

Visual effects such as fluids and smoke are volumetric in nature and thus are very difficult to visualize using geometric primitives [58]. Instead, *volumetric visualization models* can be used to render these effects.

One of the most popular volume visualization methods is *volume ray casting*. This should not be confused with standard ray casting; volume ray casting processes *volume data*, while ray casting processes *surface data* [135]. Describing the differences between these two methods will inherently explain why volume visualization is typically divided into two distinct categories.

In ray casting, geometric rays from the eye or camera find the intersection of the ray with the volume that is to be rendered. A function value is computed based on the properties of the material used to represent the volume. As the ray is essentially a sample of light, some of the light will be reflected, refracted and absorbed by the material when the ray intersects. This function value can be returned as the colour of the volume or surface at that particular point. Additionally, field functions can be used to calculate the *gradient* of the ray, which is used as the normal to the surface at that particular intersection point.

In contrast, volume ray-casting does not stop at the single intersection of the surface, and instead goes through the volume or object. Samples are taken along the ray within the volume, and the samples should be spaced apart equally. This can be achieved using a three-dimensional grid, the cells of which are typically named *voxels* (volume elements). Within each sampling point or voxel, the gradient is determined similar to the surface generation, calculating the direction of the volume surface within each particular voxel. Finally, the volume is *composited* back along the ray, effectively blending the colours of the front voxels with the colours of the voxels behind to produce a final pixel value.

Essentially, volume visualization methods can be divided into two separate categories. One category contains methods which render only the *surface* of a volume, typically called *surface rendering* or *indirect volume rendering*. The other category renders the entire volume, typically called *volume rendering* or *direct volume rendering*. Ray casting falls under the former category, while volume ray casting falls under the latter.

Volume visualization will usually approach volume data in one of two ways: a voxel or a cell approach. A voxel approach assumes the value does not change within that particular grid point. In contrast, a cell approach assumes values to vary within the cell at that particular grid point. Thus, the final value is estimated by interpolating between the values at *each corner* of

the cell. The most common interpolation is *trilinear interpolation* [62] and is the interpolation method used in this thesis.

The inherent disadvantage of volume visualization methods is that they require a large number of samples. This is not such a problem for static models or non-real-time renderings, but this can become a problem for visualization of rapidly changing data sets in real-time, or for data sets where interactivity is required. As has been discussed previously, the real-time visualization can lead to a better understanding of the data [109], so it is worth simplifying the rendering algorithms to achieve this, even if the final simulation suffers a loss of realism [143]. Luckily, volume visualization algorithms are inherently parallel, so there is plenty of room for optimization.

The volume rendering method discussed in this section is based off the method used by Green's smoke particles simulation [42], which was built on the volumetric methods described by Ikits et al. [58]. Techniques for the generation and rendering of *surfaces*, such as the popular *Marching Cubes algorithm* [73], will be explained in greater detail in Chapter 7.

The basic method of volumetric rendering is carried out in several steps:

- Calculate the half-angle vector - a vector which is halfway between the light vector and the camera eye direction.
- Sort all particles in the system *along* the half-angle vector, such that the entire volume of the system is divided into *slices*.
- Render primitives for all particles in each slice and propagate volume effects, for example shadows, back onto the slices behind.

Listing 7.5 shows the calculation of the half-angle from the camera view vector and the direction of the light. This half-angle vector is used as the sorting vector to split the volume into slices. Particles are sorted using the standard sorting algorithms of THRUST and obtain slice indices depending on which slice they belong to. Additionally, in order to draw shadows of particles from the front slices onto the back, the shadow matrix must be calculated, as shown in Listing 7.6. After transformation with the ModelViewProjection matrix, the vertices are transformed into normalized device coordinates (i.e. from $(-1, -1, -1)$ to $(1, 1, 1)$). Since texture coordinates must be between $0$ and $1$ these coordinates are converted using a "bias" matrix, basically translating the entire texture coordinates $0.5$ to the left and down.

Once these have been calculated, the only thing left to do is to render the particles similarly to other point-based methods, but one slice at a time. This rendering uses the standard OpenGL VBO rendering method of binding the vertex arrays and using `glDrawElements` along with shaders. The shaders are using the simple sprite texture shaders as per Listing 7.2. The final result is shown in Figure 7.3.

The visual results shown in 7.3 give a good effect, but it is not particularly realistic. In order to get a more realistic representation of a water *surface*, a surface generation method must be implemented. This is discussed in the next section.

```
//Calculate the sorting vector for volume rendering
float3 calc_half_vector(mats matrices, float3 light_pos){

    //Get the current model-view matrix from pipeline
    mat4 model_view = matrices.get_modelview();

    //Find light vector
    float3 light_vector = normalize(light_pos);

    //Find camera view direction
    float3 view_vector = -make_float3(model_view[0][2], model_view[1][2],
        model_view[2][2]);

    //Calculate half-angle
    float3 half_vector = normalize(view_vector + light_vector);
    return half_vector;
}
```

*Listing 7.5: Listing showing the calculation of the half-angle which is used for dividing and sorting the particles into slices. The struct $mats$ is a custom container with the values for all of the usual matrices and is updated consistently.*

```
//Calculate the sorting vector for volume rendering
mat4 calc_shadow_matrix(mats matrices, float3 light_pos){

    //Get the current model-view matrix from pipeline
    mat4 model_view = matrices.get_modelview();

    //Make light always point to the center of the simulation
    float3 light_target = make_float3(0.0f, 0.0f, 0.0f);

    //Set up light matrices
    mat4 light_view = look_at(light_pos, light_target,
                              make_float3(0.0f, 1.0f, 0.0f);
    mat4 light_proj = perspective(45.0f, 1.0f, 0.1f, 100.0f);

   //Use bias matrix to convert to NDC
   mat4 biasMatrix = mat4(
      vec4(0.5, 0.0, 0.0, 0.0),
      vec4(0.0, 0.5, 0.0, 0.0),
      vec4(0.0, 0.0, 0.5, 0.0),
      vec4(0.5, 0.5, 0.5, 1.0)
   );

   //Final shadow matrix
   mat4 shadow_matrix = biasMatrix * light_proj * light_view *
                        inverse(model_view);
   return shadow_matrix;
}
```

*Listing 7.6: Calculating the shadow matrix to be used in propagating shadows back from front slices onto back slices. The shadow matrix uses a $biasMatrix$ to transform the vertices of the shadow texture from normalized device coordinates (NDC) to texture coordinates*

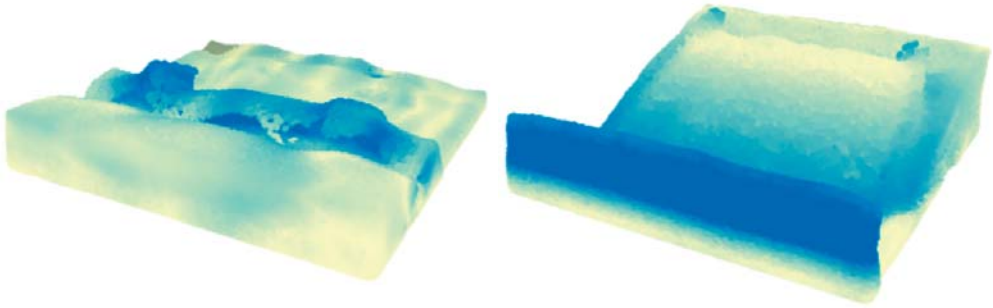

*Figure 7.3: Example rendering of the water simulation using volumetric rendering. The sprites are coloured based on the velocity of the particles.*

## 7.5 Surface Generation

So far there have been numerous methods to present particle system simulations in such a way that the rendering methods used are put on the back-burner, concentrating more on the particle system rules and behaviour while using graphics tricks and illusions to make the simulation seem realistic. While this does serve its purpose well, it begs the question - can more complex rendering methods be used to further enhance the quality of the particle system simulation?

Of course, using more complex rendering methods such as these comes at a high cost of performance. There exist many high quality simulation methods which can render large particle systems in a *post-processing* manner [74] [75] [57] [34], including the popular PBRT source code [101] for rendering very high quality images using methods specifically catering to all manner of physics-based simulations. However there are fewer methods which will be adequate for rendering high quality systems in *real-time*. Given the same amount of processing power, a more complex rendering method results in a less-complex particle system method, and vice-versa. The challenge involves balancing the two sides to achieve the most desirable outcome.

Different examples of water visualization have been approached for rendering in real-time. Kooten et al. [132] present an approach to rendering surfaces at interactive rates using a *surface particle* method, where surface particles move around on top of the particles already making up the simulation (considered as *fluid particles* to distinguish the two). Only these surface particles are used to make up the surface mesh which is rendered. Bagar et al. [5] provide a real-time rendering method using a layered-based approach similar to volumetric rendering which is particularly successful in rendering foam effects in water.

Surface generation over the top of a particle system presents a number of challenges:

- The particle system becomes *restricted* to a certain space chunk. One strength of particle

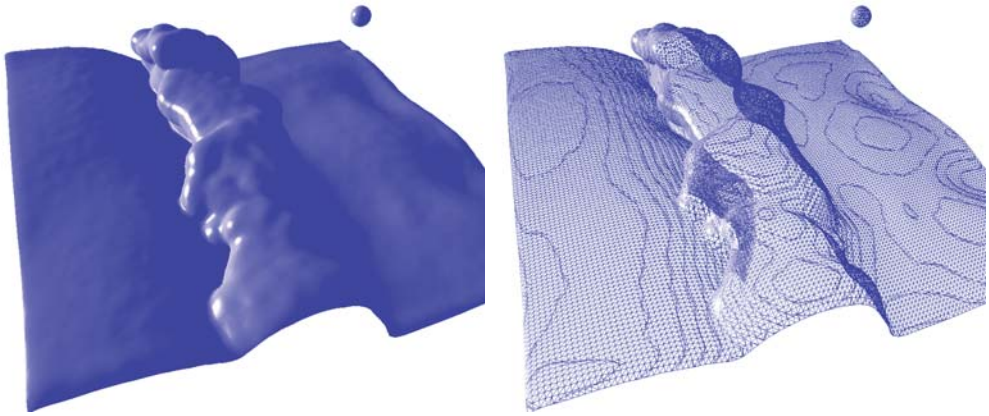

*Figure 7.4: An example of surface rendering using the Marching Cubes surface generation method. On the left, the surface has been shaded properly with a Blinn-Phong shading model. The right shows the same surface but using a wireframe model to show exactly how the individual polygons are drawn.*

systems is that they are not grid-based systems and as such, particles can move infinitely in any direction. Surfaces do not have this luxury however, so particles upon which the surface has been generated must be constrained within the boundaries where the surface can be rendered. Where appropriate, this problem can be solved by using collision detection algorithms to restrict particle movement to the same dimensions as the grid used to generate the surface. A good example of this is pouring particles into a square container. Particles cannot move outside of the container, while the surface needs only to be generated on the inside. It should be noted that other surface generation algorithms [132] remove this restriction, although the fluid particles may be contained within a certain area for simulation purposes anyway (i.e. pouring a glass of water), in which case the restriction is still applied.

- The size of the particles relative to the size of the surface grid needs to be taken into consideration. If the grid size is too large relative to the size of the particles, the system will appear to have a rough, low-quality surface because only a small number of grid elements are used to render it. This can be improved by increasing the *resolution* of the grid, however this will result in a negative impact on performance.

- Taking into account the previous point, a decrease in resolution requires an increase in the particle size. This in turn requires a decrease in the particle *number* if the system is to maintain the same volume. This is not exactly desirable, however, as the larger the number of particles in a system, the more accurate a representation of its behaviour. As always, a balance must be found between all these variables to obtain the best possible simulation.

The core concepts of an isosurface as well as the *field functions* used will first be explained in Section 7.5.1. Next, the surface generation algorithm used - the popular *Marching Cubes* algorithm [73] - will be explained in Section 7.5.2, including the general theory of the algorithm and an overview of the implementation. Finally, extra rendering techniques are introduced in Section 7.6 and optimization options are explored in Section 7.7.

### 7.5.1 Isosurface Introduction

An *isosurface* can be defined as a surface which exists between two sets of points in three-dimensional space. The two sets of points are determined by the application of a *threshold* value to some continuous function. Points *above* the threshold belong to one set (usually set to a value of $0$), while points *below* the threshold belong to the second set, usually set to a value of $1$. The continuous function used to apply the threshold to is known as the *field function* [13]. Field functions determine a value at all points in space according to some rules. Every isosurface will appear different depending on the field function used. When applying an isosurface generation method to a particle system, the first thing to consider is which field function to use to represent each particle.

Particles are most often represented by singular points or regular two-dimensional sprites as shown in the rendering methods used in Chapter 6. Moreover, since this particle system is simulating a fluid-like substance, a generically small quantity of fluid could be a *droplet* of fluid or water. Therefore, an appropriate field function to represent a particle could be a standard spherical field function, shown in the general form in Equation 7.1.

$$f(x, y, z) = x^2 + y^2 + z^2 \tag{7.1}$$

The threshold value is applied to this function by simply determining the radius of the sphere desired to represent the size of the particle. By taking into account the particle's position in three-dimensional space, a second equation for the field function is deviated based on the particle's position vector, as shown in Equation 7.2.

$$r^2 = (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 \tag{7.2}$$

where $x$, $y$, and $z$ are any point in three-dimensional space to be evaluated by the field function, $c$ is the position of the particle, a three-dimensional vector, and $r$ is the radius of the particle.

However, this field function is not particularly useful for isosurface generation when field functions are being used for *multiple* particles. As particles approach each other and/or collide, the fields will not merge as one would expect. Two spheres will simply be drawn partially inside each other. A more appropriate field function is the equation for *electric field strength of a point charge*, derived from Coulomb's law [24] (Equation 7.3) and applied to the particle system, ignoring the electrostatic charge and constants, as shown in Equation 7.4.

$$E = \frac{q}{4\pi\epsilon_0 r^2} \tag{7.3}$$

$$E = \frac{1}{(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2} \tag{7.4}$$

Basically, the strength of the field at some point $(x, y, z)$ is directly dependent on its proximity to the point charge $c$. The reason for using this equation over the one used in Equation 7.2 is that electric fields are *added*. When two particles approach each other their electric fields are added and they merge into one combined field. In computer graphics, this is what is commonly known as a *metaball* [132] [13].

The field function can be optimized further - as the distance $r$ from the center of the particle increases, the strength of the field approaches zero, but never touches. Similarly, as the evaluation point gets closer to the center of the particle, the strength approaches infinity. In order to avoid such extreme values, an approximation function can be used to more efficiently compute the field values needed. The function shown in Equation 7.5 was proposed by Triquet et. al. [130] [112] and can be used to approximate a sphere relatively well.

$$f(r) = r^4 - r^2 + 0.25 \tag{7.5}$$

where $r^2$ is given by the Equation 7.2.

Using this equation, as the evaluation point $(x, y, z)$ approaches the center of the particle (or point charge) $c$, the strength of the function approaches $0.25$. Similarly, the strength of the function equals zero when $r$ equals $\sqrt{2}/2$. Using this information, the threshold value can be set at $\sqrt{2}/2$ to produce the final isosurface. When a point has an $r$ value larger than $\sqrt{2}/2$, it is outside the sphere, and thus the field value is set to zero. The result is a spherical isosurface for each particle, which will join with other particle surfaces as they move together.

Figure 7.5 demonstrates the joining of particle fields with the other fields of nearby particles. The smallest ball (the darkest blue) represents the particle itself, with a size given by the *particle radius*. The lighter blue circle around the particle represents the particle field, with a size given by the *field radius*. This is larger than the particle radius to allow the particles to move together and join. Finally, the outer blue layer represents the final metaball, increasing in size as particle fields combine.

$$f_P = \sum_{i=1}^{N} f(r_{P,i}) \tag{7.6}$$

Equation 7.6 shows the method of combining particle fields at a particular point. $f_P$ refers to to the total field value for a given evaluation point $P$. The field function is evaluated for each

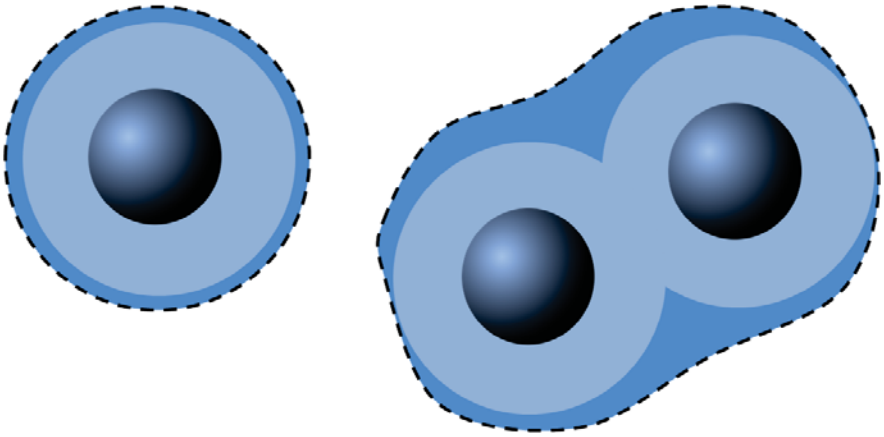

*Figure 7.5: An example diagram of particle fields joining to form metaballs. Note this is a two-dimensional image for demonstration purposes, in practice this is a three-dimensional operation.*

particle $i$, taking into account $r_{P,i}$, the distance between the evaluation point and the particle. Note that in this equation $N$ refers to the total number of particles in the local area of an evaluation point, not the total number of particles in the system.

Listing 7.7 shows the calculation of the particle field function values as a CUDA function launched on the device. The function is called by the main marching cubes kernels which will be discussed in Section 7.5.4. It takes an evaluation point $(x, y, z)$ and the center point of the particle $c$ and outputs the intensity of the field, given the radius $r$ of the *particle field*. This is not to be confused with the particle radius, which is used in fluid dynamics or physics equations in the behavioural stage of the simulation, not the rendering stage.

Specifying the correct field radius $r$ to apply to the spherical field function is one of the key aspects of setting up this rendering method to render the water system in real-time. This will be explored in more detail later, but for now simply assume that the applied field radius slightly *larger* than the particle radius used in the fluid dynamics or physics system, allowing fields to join. If the field radius is no larger than the particle radius, it will just appear like the fields are bouncing off each other and interacting just as normal particles do.

### 7.5.2 Marching Cubes

Marching Cubes is an algorithm which has seen extensive use in generating polygonal meshes from a three-dimensional scalar field since its development in 1987 by Lorensen and Cline [73]. The algorithm is most often used in computer graphics when rendering three-dimensional surfaces for modelling volumes and flows, as well as rendering medical images such as a CT scan or MRI [29]. The algorithm works first by dividing the sample space into an imaginary three-dimensional grid of cubes. When generating the polygonal surface, every corner of each cube in the grid is tested to determine which corners reside *within* the given field function and which do not. These results are then compared to a set of *predefined possible states*, calculating a configuration of triangular polygons which will best represent the surface within

```
__device__
float sphericalField(float x, float y, float z, float r, float4 c){

    float value;
    float dSq;
    float rSq = r * r;

    //Squared distance between particle pos c and eval point (x, y, z)
    dSq = ((x-c.x)*(x-c.x)) + ((y-c.y)*(y-c.y)) + ((z-c.z)*(z-c.z));

    //Final dSq takes into account the given radius
    dSq = dSq / rSq;

    //Apply optimization function
    value = (dSq * dSq) - dSq + 0.25f;

    //If ~0.707 is threshold for d, 0.5 is the threshold for dSq
    if(dSq >= 0.5f)value = 0.0f;

    //Finished
    return value;
}
```

*Listing 7.7: The particle field function. This function is called for every particle within the local area of an evaluation point (for example a marching cubes corner). For each evaluation point, field function values for every particle are summed to produce the final field value, determining how strong the total field is at that point. The more particles surrounding an evaluation point, the larger the total field strength.*

that particular cube. There are 15 base cube states, after taking into account any reflective and rotational symmetries for all possible states. These states are shown in Figure 7.6.

When the polygons from every cube are combined, the surface can be rendered. The quality of the rendering is directly related to the voxel size of the grid used to divide the field into cubes. A higher resolution grid means the cubes used to generate the surface are smaller and therefore can more accurately represent the field functions. A high resolution grid will provide a smoother surface, but because there are more cubes to test, it will take longer to generate the surface overall.

The field function used to represent the particles has already been described in Section 7.5. It is possible to use predefined 3D model values to load in a static model if the data is available, but if the data are not known or the model is not a static rendering, such as this simulation, the approximation field function can be used to generate the surface in real-time. Although the field function used for the simulation in this chapter is spherical, there is no reason why other more irregular field functions could not be used to provide some interesting surfaces. For this particular simulation only the spherical field function will be used.

Figure 7.7 shows a more detailed diagram of the process of determining the Marching Cubes state of any given cube by the arrangement of the particle fields around and inside the cube. Here, four fluid particles are within area where it may be possible for them to influence the marching cube state. The particle fields on the extreme left and right, which are coloured slightly more grey than the others, are too far away or simply do not contain the corners of

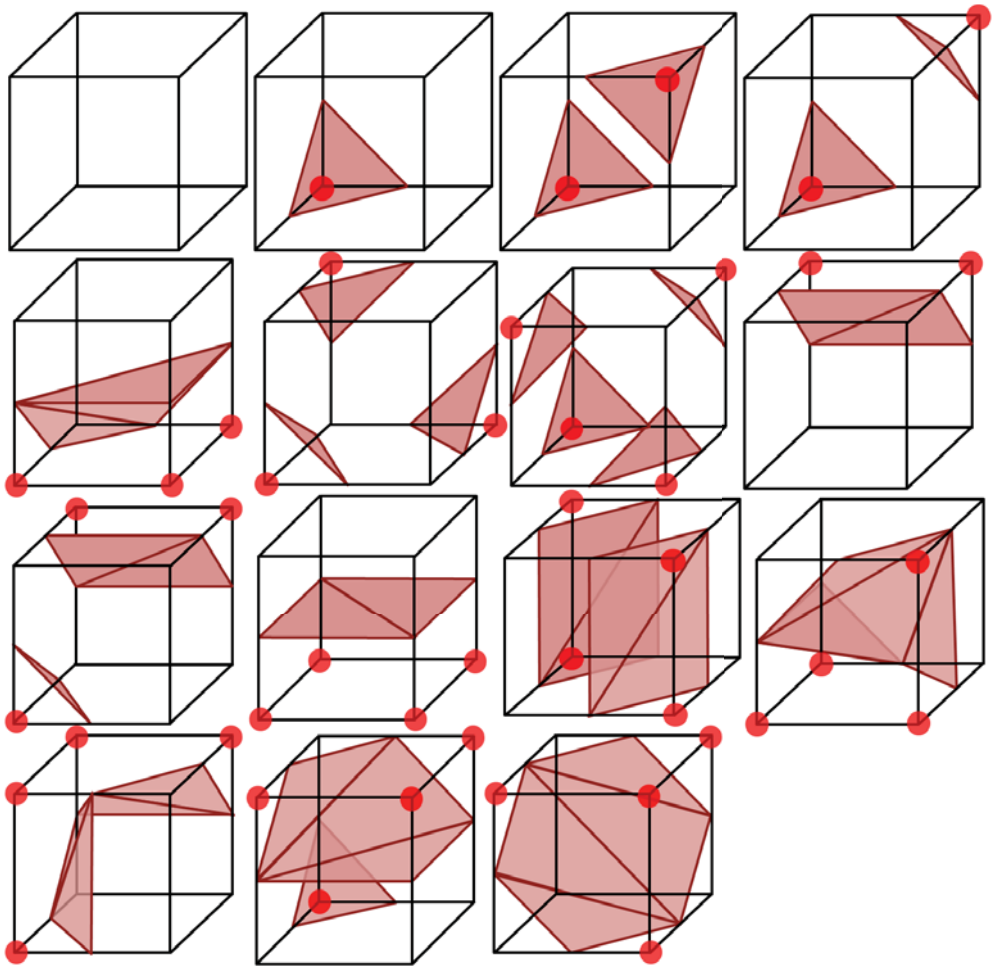

*Figure 7.6: All base Marching Cubes states. These states may be rotated and flipped to obtain the total 256 possible states for the polygons. Red dots indicate which corners are considered to be within the surface. From this angle it is sometimes difficult to judge which side of the polygon refers to the outer side of the surface - as a general rule, the normal to the surface of the polygon should always point away from the nearest red corner point.*

the cube so they are ultimately ignored. Instead, these will be used to define the state of other neighbouring cubes where they have greater influence. The larger metaball in the middle is big enough to contain the top back two corners of the cube so this is what determines the final Marching Cube state and consequently the polygon configuration of this cube, as shown on the right hand side of the diagram.

The Marching Cubes algorithm lends itself well to parallelization, with the Marching Cubes grid able to be divided efficiently into thread blocks for concurrent execution. Each cube "marching" through the field can be run on a single thread, making for significant performance improvements in larger grid configurations. The Marching Cubes algorithm can be broken down into several stages:

1. Retrieve particle data from the particle system.

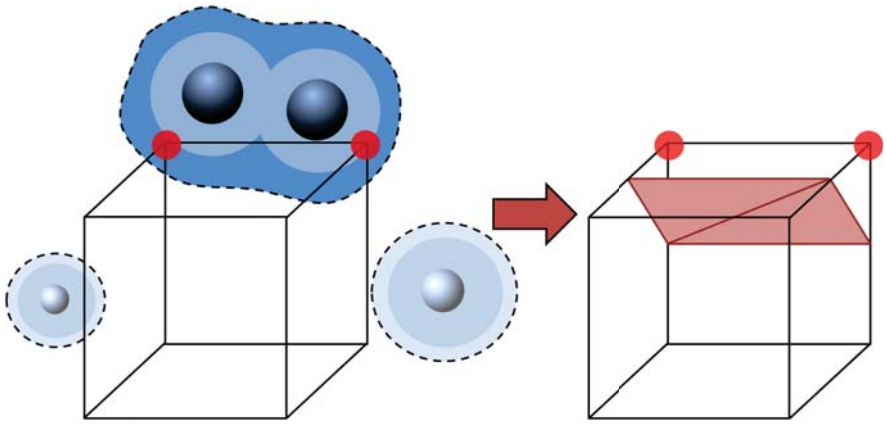

*Figure 7.7: Diagram of the particle fields influencing the marching cube state for a particular cube in the grid. Corners marked as inside the surface are determined by the surrounding particle fields, and from here the resulting polygons are calculated based on the marching cube state.*

2. Classify Marching Cubes cells according to particle positions.

3. Remove all empty cells.

4. Generate all geometric data, including surface vertices and normals.

5. Render the surface.

There is a more in-depth explanation these stages in the following subsections. Subsection 7.5.3 describes how the Marching Cubes algorithm is initialized and subsection 7.5.4 describes the general overview of the kernel. Subsection 7.5.5 describes the different cell classification kernel approaches, while Subsection 7.5.6 covers the kernel used to generate the surface.

### 7.5.3 Initialization

The area wherein Marching Cubes will generate the surface is initialized into a three-dimensional grid whereby each cube in the grid is assigned to one CUDA thread. As usual, parallelization is best done in powers of two, however the quality of the generated surface must also be considered when determining the dimensions of the grid. If it is not large enough, the surface will be blocky with relatively few polygons. A larger grid will create a smoother surface, but too many polygons will cause the system to render slowly, regardless of the parallelization speed-up.

The vertices of every cell are stored in the main integer vertex array copied into global memory. Given the cell ID calculated at the beginning of the kernel, this array is used to find the position of the current cell inside the grid as well as the positions of the other seven vertices, which will be used in the Marching Cubes algorithm. One step of the cell length in each of the $x$, $y$ and $z$ directions can find these points. Arrays also need to be initialized for sorting the marching cube data (see Section 7.5.4).

Additionally, the core of the Marching Cubes algorithm requires the usage of *three lookup tables* for comparison of the 15 possible cube states [73]. These three lookup tables include a table for *edge intersection* with the current cube, a *triangle map* to determine the triangle configuration of the calculated state, as well as a *vertex table* to determine which vertices will be drawn for the aforementioned triangle configuration.

### 7.5.4 Implementation Overview

The Marching Cubes algorithm is divided up into two main kernels and three additional kernels used primarily for sorting the Marching Cubes data. The vast majority of the computation time (including particle kernels) is used by the two main kernels. The first kernel `classifyCell` calculates which Marching Cubes grid cells are empty and appoints them to be removed from the surface generation stage of the algorithm. The second kernel `generateTriangles` is for actually generating the vertices used to render the surface.

Between these two kernels, the state of the particle system at the time will dictate which kernel will need to work harder, as there will be relatively few cells needing to be removed from the algorithm if the particles are spread in a uniform distribution in the grid space. Consequently, a greater number vertices will need to be calculated. In contrast, if particles are gathered in one particular place (for example, a surface of water would gather particles toward the lower area of the grid space, leaving the upper spaces empty) the `classifyCell` kernel works much harder as there are far more empty cells to remove. Of course, this all depends on the particle system that is being simulated. Relatively high particle attraction (due to large viscosity, for example) as well as a large global gravitational force, will typically leave a large number of cells empty. The raw number of particles can also make a big difference. If the render space is almost filled up with particles, many cells will be removed because cells completely *inside* the surface or mesh (i.e. Marching Cubes State 15) do not contribute to the surface generation, so they can also be classified as empty as well.

Once the empty cells are removed, concurrent threads are not wasted when generating the final surface in the second kernel `generateTriangles`. To remove the classified empty cells, a thrust `exclusive_scan` is performed to sort the cells. This allows empty ones to be removed by an additional `removeEmpty` kernel as well as the total number of *occupied cells* to be counted. The total number of *vertices* contained in each of those cells is also calculated. All this information can be used in the `generateTriangles` kernel to generate the final surface significantly faster than it would without any cell classification.

### 7.5.5 Cell Classification Approaches

In order to determine which cells should be removed, the `classifyCell` kernel performs a *test-run* of the surface generation kernel. This involves taking each particle's position and comparing it to the current Marching Cubes cell position to determine if that particle's field intersects with the current cell, thus forming part of the surface. This can be done in two ways, a naive approach and a grid-based sorting approach.

```
//Position of the marching cubes grid cell
float3 cell_pos;

//Evaluate at each corner
float field[8];
for(int i=0; i<8; i++){
   field[i] = 0.0f;
}

//Evaluate for all n particles
for(int i=0; i<n; i++){
    float4 particle = particles[i];
    field[0] += particleField(cell_pos, particle);
    field[1] += particleField(cell_pos +
      make_float3(cell_size.x, 0, 0), particle);
    field[2] += particleField(cell_pos +
      make_float3(cell_size.x, cell_size.y, 0), particle);
    field[3] += particleField(cell_pos +
      make_float3(0, cell_size.y, 0), particle);
    field[4] += particleField(cell_pos +
      make_float3(0, 0, cell_size.z), particle);
    field[5] += particleField(cell_pos +
      make_float3(cell_size.x, 0, cell_size.z), particle);
    field[6] += particleField(cell_pos +
      make_float3(cell_size.x, cell_size.y, cell_size.z), particle);
    field[7] += particleField(cell_pos +
      make_float3(0, cell_size.y, cell_size.z), particle);
}
```

*Listing 7.8: Classifying cells - the naive approach*

The naive approach to this problem lets each Marching Cubes cell check each individual particle. The particle's position is compared to each corner of the Marching Cubes cell to determine which corners exist inside the field of the particle. This method is shown is Listing 7.8.

The function `particleField` calls the spherical field function that was explained in Section 7.5. `cell_size` is a constant variable which represents the size of each Marching Cubes cell. Once the field values have been finalized, the algorithm will use them to find which marching cube state best represents the current cell (this will be described later).

The obvious problem here is that the computational complexity scales exponentially as the number of particles increases. Unlike the general method of parallelization from Chapter 6 where the system is parallelized one particle per thread or one fluid grid cell per thread, the algorithm is parallelized at the Marching Cubes cell level. This presents the inherent problem in combining these two methods; which method should be prioritized for parallelization? Parallelizing the particles instead would not solve the problem, as each particle could be required to compare with many millions of Marching Cubes cells (in a high-resolution simulation, as described later in this thesis) and the complexity issue would be even more noticeable. However, parallelizing Marching Cubes introduces problems when the number of particles in the system becomes large (as seen here), and for highly accurate and realistic simulations, this is usually the case.

The solution is to employ a grid-based sorting method similar to the one used to parallelize the particles described in Chapter 5. The method is very similar to the spatial grid implementations for the methods of interacting particles for the fire and water systems. Only particles close to the world position of the current marching cube cell are used in the calculation of the field values for each corner of the cell. This operates under the premise that particle fields far away from the cell have zero effect on the field values at each corner of the cell. Listing 7.9 demonstrates this method.

As shown in the listing, the code is very familiar to the original ”naive approach” with the fields for each corner calculated using the `particleField` function. Here the particle cell size values have been abbreviated to `cs` for brevity. Indices into the *first* and *last* arrays are calculated using the standard means (as explained in earlier chapters) and these are what produces a final list of neighbouring particles which will possibly influence the field values for the marching cube cell corners. For each of the neighbouring cells in the grid, the particles contained within these cells are checked using the `particleField` function as to whether they affect the field values for each corner of the Marching Cubes cell. Note that this is dependent on the *world position* of the cell, *not* the position of the cell within the spatial grid. Otherwise the rest of the method remains basically the same and the result will be an array of values for each corner of the cube which will determine the final Marching Cubes state for that cell.

It is important to note that the spatial grid indexing used for this Marching Cubes phase of the simulation is completely *different* from the spatial grid indexing used in the behavioural phase. This is very important because the two grids used in each phase are different sizes and therefore will contain different lists of particles in each cell. For the behavioural phase, the cell size is determined by the particle radius (or for SPH, the smoothing kernel size), but the cell size for the *Marching Cubes* phase is determined by the particle *field radius*. Ultimately this produces a situation similar to the spatial grid implementation used for the simple *plasma* simulation - the field radius is always larger than the particle radius (or smoothing kernel size) so there will be many more particles inside each Marching Cubes spatial grid cell than would otherwise be the most efficient (i.e. one particle per cell). This is unavoidable as we are requiring particle fields to merge and join to create the surface - it stands to reason that the metaball resulting in the merging of two fields from particles very close together will have both particles contained within the same Marching Cubes spatial grid cell if the grid cell is the size of the standard particle field radius.

Additionally, it is also important to distinguish the Marching Cubes spatial grid from the *Marching Cubes grid* - i.e. the grid of cells which is being parallelized. The latter has a much higher resolution than the former. For instance, consider how polygons could be laid out to render a sphere - one method would be to draw vertical triangle strips around some axis of the sphere. Each one of those triangles needs to be drawn within a single Marching Cubes cell (based on the cube state), so it is impossible to draw a single sphere within one Marching Cubes cell. Indeed, it would take many Marching Cubes cells to draw this sphere and this number increases as the sphere surface becomes smoother. A representation of a sphere at the *smallest* resolution would be a double-pyramid representation (two pyramids joined at the base) which

```
//Find position in spatial grid cell from world position
float3 world_pos;
int3 cell = calcCellPos(world_pos);

//Evaluate for 8 corners
float field[8];
for(int i=0; i<8; i++){
   field[i] = 0.0f;
}

//Find only particles whose fields will affect the current cell
for(int z=-1; z<=1; z++){
  for(int y=-1; y<=1; y++){
      for(int x=-1; x<=1; x++){

          //Neighbour is current cell when x,y,z = 0
          int3 neighbour = cell + make_int3(x, y, z);
          uint cell_index = calcCellIndex(neighbour);
          uint first_index = cell_first[cell_index];

          //Cell is not empty (cells are memset initially)
          if(first_index != 0xffffffff){

              //Only fields in the immediate area effect the MC cell
              uint last_index = cell_end[cell_index];
              for(int i=first_index; i<last_index; i++){
                  float4 particle = particles[i];
                  field[0] += particleField(world_pos, particle);
                  field[1] += particleField(world_pos +
                    make_float3(cs.x, 0, 0), particle);
                  field[2] += particleField(world_pos +
                    make_float3(cs.x, cs.y, 0), particle);
                  field[3] += particleField(world_pos +
                    make_float3(0, cs.y, 0), particle);
                  field[4] += particleField(world_pos +
                    make_float3(0, 0, cs.z), particle);
                  field[5] += particleField(world_pos +
                    make_float3(cs.x, 0, cs.z), particle);
                  field[6] += particleField(world_pos +
                    make_float3(cs.x, cs.y, cs.z), particle);
                  field[7] += particleField(world_pos +
                    make_float3(0, cs.y, cs.z), particle);
} } } } }
```

*Listing 7.9: Classifying cells - spatial grid based method*

is made up of eight Marching Cubes grid cells, each with only one corner contained within the field. Examples of the differences in Marching Cubes grid resolution are explained later in the chapter.

Once the field values have been calculated, an index into the Marching Cubes vertices lookup table is calculated based on the set *isovalue*, as shown in Listing 7.10. The isovalue determines the final state of the cell corner given its field value. If the field value is less than the isovalue, the corner is considered to be *inside* the surface, while a greater value is considered *outside*. The final `cubeindex` can be calculated by adding each of these corner states. The number of vertices is then found using the Marching Cubes vertex lookup table, containing the

```
//Calculate flag indicating if each vertex is inside or outside isosurface
uint cubeindex;
cubeindex =  uint(field[0] < isovalue);
cubeindex += uint(field[1] < isovalue)*2;
cubeindex += uint(field[2] < isovalue)*4;
cubeindex += uint(field[3] < isovalue)*8;
cubeindex += uint(field[4] < isovalue)*16;
cubeindex += uint(field[5] < isovalue)*32;
cubeindex += uint(field[6] < isovalue)*64;
cubeindex += uint(field[7] < isovalue)*128;

//Read number of vertices from lookup tables
uint nVertices = verticesLookup[cubeindex];

//Return number of vertices and empty/occupied
if (i < numCells){
   cell_vertices[i] = nVertices;
   cell_occupied[i] = (nVertices > 0);
}
```

*Listing 7.10: Looking up Marching Cubes state*

number of vertices to be rendered for a Marching Cubes state specified by `cubeIndex`. If the cell contains a number of vertices, the value is stored in the `cellVertices` array. If it has zero vertices, the cell is marked empty and will later be removed from the surface generation algorithm when the arrays are sorted.

### 7.5.6 Generating the Surface

With the classification and sorting stage of the algorithm complete, an array of cells has been produced which will contain some part of the surface needed to be generated. The surface generation kernel will run for each of these cells to determine the exact polygon composition needing to be rendered in order to produce the final surface.

There is some code duplication from the cell classification kernel. This is not surprising as the cell classification kernel was designed to be a test-run of this one. The main difference is that the surface generation kernel must also calculate the normals to the vertices. This requires a secondary step once the field values have been calculated - the gradients must be calculated based on the field values from each corner of the cube. The code used to calculate the field values and gradients for each corner of the Marching Cubes cell is shown here in Listing 7.11.

This listing has been altered from its original code to omit repetitive code which is unavoidable in this instance. Firstly, the spatial grid calculation is performed as described in Listing 7.9 and a list of possible influencing particles is made. The normal gradient is then calculated based on the field value for each corner of the cube using the `fieldGradient` function. The way this works is essentially a *test point* is created a set distance outside from the cube corner, and then the field value is calculated again using this test point. If the resulting field value is *greater* than the value provided by the actual cube corner, it means that the test point is *closer* to the center of the field - since particle fields are spherical, the center of the sphere must then

```
//Final field made up of normal (float3) and value (float)
float4 field[8];
float field_value[8];
float3 gradient[8];
float temp = 0.0f;

//Duplicate spatial grid code omitted for brevity
//Iterating though local particle list...
    float4 particle = particles[i];

    //Gradient at each corner is determined by values at all eight corners
    for(int i=0;i<8; i++){
        //Assume corner world positions are pre-calculated
        temp = particleField(corner[i], particle);
        field_value[i] += temp;
        gradient[i] += fieldGradient(corner[i], temp, particle);
    }

for(int j=0; j<8; j++){
   field[j] = make_float4(gradient[j], field_value[j]);
}

//cubeindex calculation same as in the above listing
//...
//Now output triangle vertices
uint nVertices = verticesLookup[cubeindex];
for(int i=0; i<nVertices; i++){
    uint edge = trianglesLookup[cubeindex*16 + i];
    uint index = cellVertices[cell] + i;
    pos[index] = make_float4(vertices[edge], 1.0f); //Output vertex position
    norm[index] = make_float4(normals[edge], 0.0f); //Output vertex normal
}
```

*Listing 7.11: Calculating the particle field value and gradient*

be more towards outside of the cube than inside (if the test point is greater). Therefore, the surface normal should be pointing towards the inside of the cube corner in this case. On the other hand, if the evaluation at the test point returns a value smaller than that at the corresponding corner, the field center will be more *inside* of the cube than outside, and therefore the normal should point *away* from the cube corner. This is performed eight times for each corner until the final normals and field values have been reached.

Next, the cube index is calculated as per Listing 7.10 and the `vertices` and `normals` arrays for each cube cell have been pre-calculated using a standard trilinear interpolation of the various cell corner configurations. From here it is just a matter of using the Marching Cubes lookup tables to find the final vertex and normal configuration for the polygons representing the final marching cube state for this particular cube. This data is finally stored in the VBO arrays `pos` and `norm`. To be clear, the array `pos` does not represent the positions of the particles, rather, the positions of all the vertices making up the final surface mesh.

## 7.6 Surface Rendering

The rendering of this simulation can be split into two distinct parts; The first is the core rendering of the surface using the output data from the Marching Cubes algorithm. This will be covered in Section 7.6.1. The second part is an optional step where the surface is then shaded using GLSL to look like realistic water. This will be covered in Section 7.6.2.

### 7.6.1 Surface Rendering

OpenGL-CUDA interoperability is used to render the surface quickly without having to transfer rendering data from the device to the host. Unlike basic point-based rendering methods described in Chapter 6, the particle's position data is not mapped to the vertex buffer object (VBO). Instead, the output from the `generateTriangles` kernels is mapped; both the vertex data and the normal data.

Although the vertices and the normals of the surface have been provided, it has not yet been specified how to actually render the surface. One of the most common ways of rendering a surface is to use a simple phong shading model [102]. The phong shading model is used often in computer graphics and is a simple but effective way to demonstrate the base structure of a surface. It uses a linear interpolation method to approximate the normal of the surface making it smooth. A light source can be added for illumination - the position of the light can be passed as a parameter to the GLSL shader, but in practice the *colour* of the light can be hardcoded in as we can assume ambient hitting the surface is white light (colour values will be multiplied by the skybox pixel values later).

The shading model used for this particular surface is the Blinn-Phong shading model [12], a modification on the standard phong shading model which is less computationally expensive. The vertex and fragment shaders are given in greater detail in Listings 7.12 and 7.13. Notice that the code has been simplified somewhat by hardcoding light and material properties. The shader is designed to allow the directional light to be moved around on the fly and a base colour for the surface to be provided via the vertex shader. The final result of applying this shader to the surface generated by a water simulation is shown in Figure 7.8. However, this is not the end of the rendering method. Phong shading models are great but to get a really realistic looking rendering, proper reflection and refraction must be implemented.

```
//Matrix transformations also passed as uniform variables
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 colour;
out vec3 v_eye;
out vec3 v_world;
out vec3 v_normal;

void main(){
    vec4 position = vec4(vertex, 1.0);
    gl_Position = ModelViewProjectionMatrix * position;
    v_eye = vec3(ModelViewMatrix * position);
    v_world = vec3(ModelMatrix * position);
    v_normal = NormalMatrix * normal;
}
```

*Listing 7.12: GLSL vertex shader for the Blinn-Phong shading model*

```
//Light properties(ambient/diffuse/specular/position)
uniform vec3 l_amb;
uniform vec3 l_dif;
uniform vec3 l_spe;
uniform vec3 l_world;  //Light position in world space
uniform float spe_exp; //Specular exponent

//Material properties(ambient/diffuse/specular)
uniform vec3 m_amb;
uniform vec3 m_dif;
uniform vec3 m_spe;

//From vertex shader
in vec3 v_eye;
in vec3 v_world;
in vec3 v_normal;

layout (location = 0) out vec4 out_colour;
void main(){
    vec3 normal = normalize(v_normal);              //Normal dir
    vec3 to_cam = normalize(-v_eye);                //View dir
    vec3 to_light = normalize(l_world - v_world); //Light dir

    //Ambient intensity
    vec3 i_amb = l_amb * m_amb;

    //Diffuse intensity
    float diffuse = max(dot(to_light, normal), 0.0);
    vec3 i_dif = l_dif * m_dif * diffuse;

    //Specular intensity
    vec3 half = normalize(to_cam + to_light); //Blinn-phong uses half
    float specular = max(dot(half, normal), 0.0);
    specular = pow(specular, spe_exp);
    vec3 i_spe = l_spe * m_spe * specular;

    //Note: Assume no emission colour
    out_colour = vec4(i_spe + i_dif + i_amb, 1.0);
}
```

*Listing 7.13: GLSL fragment shader for the Blinn-Phong shading model*

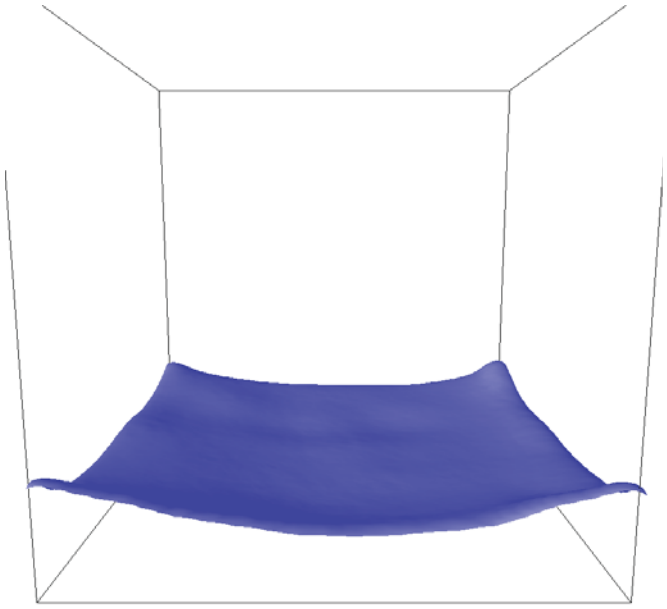

*Figure 7.8: Example result of rendering a water particle system surface using the Blinn-Phong shading model.*

### 7.6.2 Cubemapping and Fresnel Reflection

The rendering of a fluid substance is not always straightforward. Translucent objects and fluids such as water allow light to pass through them. Thus the colour of a point on the surface of the fluid is a combination of the light which reflects off the surface as well as the light which passes through. This is known as *reflection* and *refraction*.

A surface is an interface between two different substances [1]. For instance, when simulating water, the two substances are water and air. When light hits the surface it will refract differently depending on the properties of the two materials. The interaction between these the two substances is determined by the *Fresnel Equations* [103]. The Fresnel Equations describe how much light reflects off the surface, depending on the angle of incidence $\theta_i$, the *polarization* of the light (either perpendicular to the surface, or parallel to it), and finally the *index of refraction* of each of the materials. The index of refraction is typically written as $n_i$ and $n_t$ referring to the index of refraction of the incident and the refracting (transmitting) material respectively. The relationship between the indices of refraction and the angle of incidence is given by Snell's Law [53] [141] or the law of refraction, as shown in Equation 7.7. The law of reflection [53] (Equation 7.8) states that the angle of incidence $\theta_i$ and the angle of reflection $\theta_r$ must be the same.

$$n_i \sin\theta_i = n_t \sin\theta_t \tag{7.7}$$

$$\theta_i = \theta_r \tag{7.8}$$

How much light is reflected is called the *reflectance* and is dependent upon the polarization of the light, given by Equation 7.9. This equation assumes the light to be unpolarized, or containing equal parts of parallel- and perpendicular polarized light. Given the separate Fresnel equations for parallel and perpendicular polarized light [53], the final equation for the total reflectance of light at some angle of incidence $\theta_i$ is given by Equation 7.10 [141]. These equations assume that the reflection is an *external reflection* - the refraction index of the incident medium must be less than that of the transmission medium. This is the case for this simulation, as water is more dense than air and thus it has a higher refraction index.

$$R_\theta = \frac{R_\parallel + R_\perp}{2} \tag{7.9}$$

$$R_\theta = \frac{1}{2}\left(\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\right)\left(1 + \frac{\cos^2(\theta_i + \theta_t)}{\cos^2(\theta_i - \theta_t)}\right) \tag{7.10}$$

Applying the Fresnel equations to the lighting of a surface results in the reflectance increasing as the angle of incidence $\theta_i$ increases, becoming $1$ when the angle of incidence is completely perpendicular to the surface. On the other hand, as $\theta_i$ decreases to $0$ (normal incidence), the reflectance also decreases until the light is completely transmitted [1]. As such, the equation $R + T = 1$ can be used to describe the relationship of the total reflectance to the total transmittance, or in simpler terms, the reflection and refraction co-efficients. Figure 7.9 describes the process of Fresnel light modelling.

Because the Fresnel formulas are relatively complicated, it is common to use approximations for them. Schlick [115] proposed an approximation which is commonly used, as shown in Equation 7.11. It simplifies the equations by providing an approximation of the value of $R_\theta$ which is reasonably accurate for most materials [1]. The value of $R_0$ is calculated using the refractive indices of the two materials, as shown in Equation 7.12.

$$R_\theta \approx R_0 + (1 - R_0)(1 - \cos\theta_i)^5 \tag{7.11}$$

$$R_0 = \left(\frac{n_t - n_i}{n_t + n_i}\right)^2 \tag{7.12}$$

Implementing reflections and refractions onto a surface can be done relatively easily in OpenGL and GLSL once the set of equations have been decided upon. A common approach is using *cube mapping* or using a *skybox*. The idea is to have the surface surrounded by six textures arranged in a box shape, and to project the colours of the skybox onto the surface as the reflection colours. It is also possible to map textures to a surrounding *sphere* instead

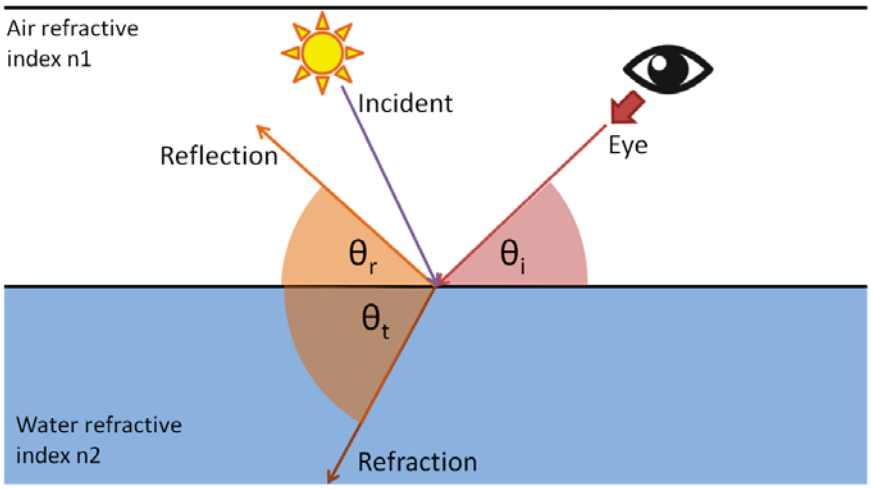


*Figure 7.9: Diagram of the way light rays interact with the surface using Fresnel light modelling. Incident rays hit the surface and produce a reflected ray and a refracted ray at different angles. In the implementation, both these rays point to different colour values on the surrounding skybox. As the angle of the ray changes, these reflection and refraction angles also change.*

of a box for a slightly more realistic approach, but this not usually noticeable. The box also serves as the texture lookup for the refraction - thus a typical skybox might have five of the six textures representing the sky and the horizon in all directions, while the sixth would represent the ground, which could be seen clearly if looking at the surface it directly from above (almost completely refracted).

OpenGL performs cube mapping by binding a texture image to a preset texture type, `GL_TEXTURE_CUBE_MAP`, a set of six textures to serve as the skybox. `samplerCube` is used by GLSL to access the cube map texture data when looking up the pixels. The vertex shader for the Fresnel rendering is almost identical to the one listed in Listing 7.12, the only difference is that the base colour is not defined by an input colour, rather it is defined purely by the samplerCube input in the fragment shader. This fragment shader is shown in Listing 7.14, and assumes vertex positions and normals are already converted into eye-space by the vertex shader. This shader for true Fresnel reflections has adapted from [141] for GLSL, while Listing 7.15 shows the fragment shader for the Schlick approximation approach [115].

The shader calculates the reflection and refraction vectors using the GLSL functions `reflect` and `refract`, taking into account the refractive indices `n1` (air) and `n2` (water). When using cube mapping, it is assumed that light is hitting the surface from every direction, thus the incident light vector is provided by the normalized camera position in eye-space reflected around the surface normal. Anything which is not reflected is refracted, therefore the refracted co-efficient equals one minus the reflected co-efficient provided by the Fresnel formula. Since the cube map exists in world-space, the reflection and refraction vectors must first be converted to world-space, which can be done easily by multiplying by the inverse of the view matrix. The texture lookup is then performed using these vectors to get the final reflection

```
uniform mat4 ViewMatrix;
uniform mat4 ViewInverse;
uniform samplerCube cubeMap;
in vec3 v_eye;
in vec3 v_normal;
layout (location = 0) out vec4 out_colour;

void main(){
    vec3 incident = normalize(v_eye);
    vec3 normal = normalize(v_normal);

    //Refractive indices
    float n1 = 1.0;         //Air
    float n2 = 1.3333;      //Water

    //Calculate the reflected and refracted vectors
    vec3 reflected = reflect(incident, normal);
    vec3 refracted = refract(incident, normal, (n1 / n2));

    //Fresnel calculations
    float ri = n1 / n2;
    float c = dot(normal, reflected) * ri;
    float g = sqrt( 1.0 + (c * c) - (ri * ri) );

    float fresnel = 0.5 * ((g-c)/(g+c)) * ((g-c)/(g+c)) *
    (1 + ( ( (c*(g+c) - (ri * ri)) / (c*(g-c) + (ri * ri)) ) *
    ((c*(g+c) - (ri * ri)) / (c*(g-c) + (ri * ri))) ) );

    float reflected_coeff = fresnel;
    float refracted_coeff = 1 - fresnel;

    //For cubemap lookup, need to convert back to world-space
    reflected = vec3(ViewInverse * vec4(reflected, 0.0));
    refracted = vec3(ViewInverse * vec4(refracted, 0.0));

    vec3 reflected_colour = vec3(texture(cubeMap, reflected));
    vec3 refracted_colour = vec3(texture(cubeMap, refracted));

    out_colour = vec4( (reflected_colour * reflected_coeff) +
                       (refracted_colour * refracted_coeff), 1.0);
}
```

*Listing 7.14: Fresnel Fragment Shader*

and refraction colours. These colours are multiplied by their respective Fresnel coefficients and then combined to get the final fragment colour.

Figure 7.10 shows the result of these shaders applied to the surface. There is little difference between the true Fresnel and the Schlick approximation shaders, however both look fairly realistic.

At the top left, there is a screenshot of the Fresnel shader without the visible skybox in the background. It makes it difficult to see the proper reflections without seeing the surrounding environment, so the following screenshots are given with the skybox visible. The top-right is a similar screenshot except with the surrounding skybox also rendered, which allowing a clearer view of the reflection of the clouds in the water as well as the refraction into the dark grey

```
uniform mat4 ViewMatrix;
uniform mat4 ViewInverse;
uniform samplerCube cubeMap;
in vec3 v_eye;
in vec3 v_normal;
layout (location = 0) out vec4 out_colour;

void main(){
    //Same as true fresnel shader
    ...
    //Schlick's approximation
    float R_0 = (n1 - n2) / (n1 + n2);
    R_0 *= R_0;
    vec3 cos_theta = dot(normal, reflected);
    float fresnel = R_0 + (1 - R_0) * pow(1 - cos_theta, 5.0);
    float reflected_coeff = fresnel;
    float refracted_coeff = 1 - fresnel;

    //Texture lookups same as true fresnel shader
    ...
    out_colour = vec4( (reflected_colour * reflected_coeff) +
                       (refracted_colour * refracted_coeff), 1.0);
}
```

*Listing 7.15: Schlick Approximation Shader*

coming from the bottom of the skybox.

The screenshot in the bottom-right shows a closeup of a turbulent fluid scenario, as particles are initialized randomly and dropped into the center of the simulation space. This provides a good example of the droplet effect as particles escape while splashing, showing the water with the reflection/refraction effect of the Fresnel. The bottom-left shows a wave-like scenario, where particles are initialized at one end of the simulation space and flow down towards the other end. The skyboxes used here are courtesy of OpenGameArt [98].

It is also interesting to consider the effects when simulating the interaction between the fluid and rigid bodies, or simply imposing boundary conditions on the fluid (not including the standard container). This is shown in Figure 7.11. Two different rigid-body interactions are performed - firstly an additional *collider particle* is used to dynamically move across the fluid and influence the flow. This is shown in the image on the left. Secondly, the fluid is dropped through a hole at the top of the simulation space and flows down until it hits the bottom of the container. This is shown in the right image, where the actual rigid-body wall has been made invisible to better see the flow of the fluid.

It should be noted that, especially in the case of the interaction with the collider particle, *real* Fresnel reflection should *include the collider in the reflection*. This is what is known as *dynamic reflection mapping* and it a bit more complicated than what has been used in the Fresnel implementation in this thesis. This implementation simply draws pixels straight from the skybox textures and ignores any objects in the way.

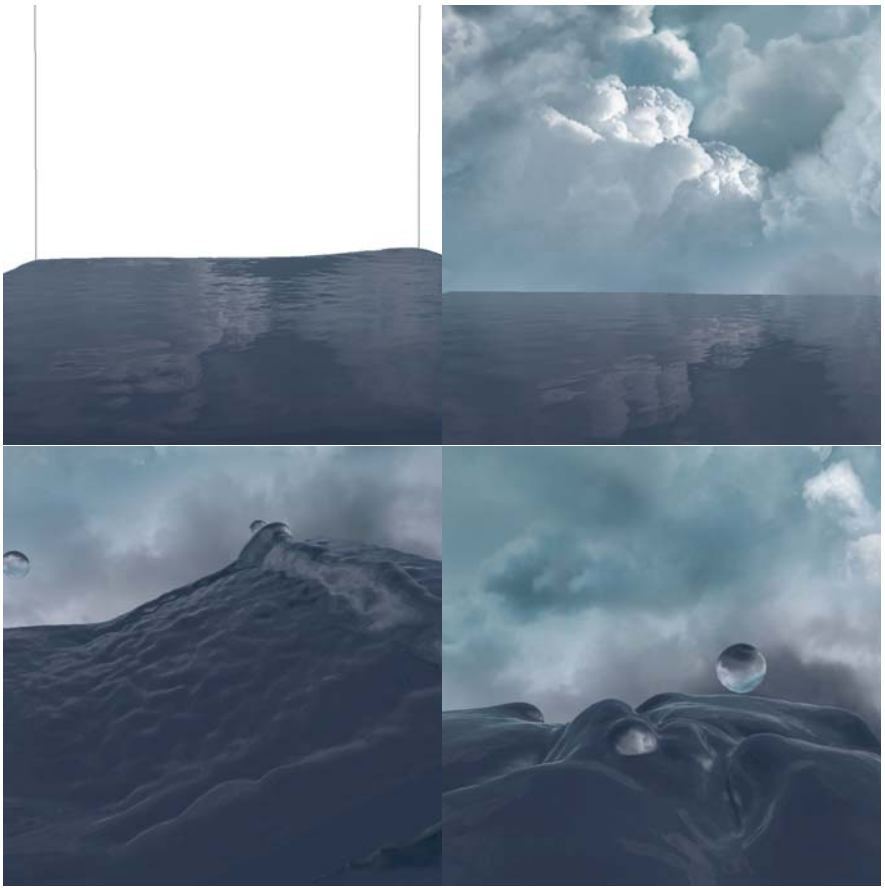

*Figure 7.10: Different fluid shading using the Fresnel effect. Each screenshot is designed to demonstrate a different effect of the rendering method. Top left: surface without skybox visible. Top right: Full reflection of clouds on a calm fluid. Bottom left: Wave effect. Bottom right: Droplet effect in turbulent fluid.*

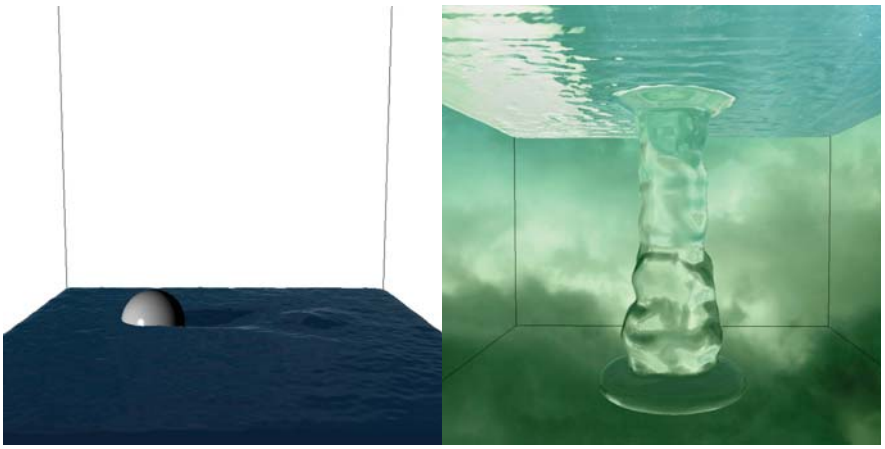

*Figure 7.11: Different interactive scenarios. Left: Interaction with collider. Right: Flow through hole in the ceiling.*

## 7.7 Results Discussion

In the introduction section, there were several points brought up as challenges to the combination of a particle system and a surface generation method. When testing this method there arise several different variables which greatly affect how well the program performs as well as how well the surface renders. These are:

- The particle system size.
- The Marching Cubes grid size.
- The particle field radius size.

When obtaining performance results for the surface generation, these three aspects in particular are considered. In general, a larger particle system size results in a slower simulation overall (this is already known from previous tests), but this is amplified in relation to the size of the particle field - large fields influence more marching cube cells, thus more fields due to more particles slow the performance down even further. Particle field size on its own must be considered from a visual perspective - particles should not be too large and make the fluid seem too "globby". Additionally, the field radius in relation to the Marching Cubes grid size also must be considered - if the field size is too small (in relation), the resulting Marching Cubes states will give "sharp" corners to what would otherwise be a smooth sphere.

In terms of units, the field radius is defined relative to the total size of the simulation area. For the results in this section, the simulation area used was a cube ranging from $-0.5$ in the $x$, $y$ and $z$ directions, to $+0.5$ in the $x$, $y$ and $z$ directions. Therefore the cube in which the surface is rendered is a $1 \times 1 \times 1$ centered at *(0, 0)*. This area is divided up equally into Marching Cubes voxels of a certain size depending on the resolution of the Marching Cubes grid (number of voxels), and the field radius size is set at a fraction of this simulation area. This way, if the simulation area must be increased (perhaps to accommodate a particular 3d model or scene) the field radius will be scaled accordingly, while keeping the same Marching Cubes grid resolution.

Tables 7.1 and 7.2 show the test results for different scenarios changing these three attributes. The results are measured in frames per second (FPS), which means the times include both the computational aspects of both the particle model *and* the surface generation, and also

| Field Radius | 32k Particles | 64k Particles | 128k Particles |
|---|---|---|---|
| 0.0625 | 34.4 | 17.8 | 9.7 |
| 0.0313 | 99.7 | 55.9 | 34.7 |
| 0.0156 | 158.9 | 103.6 | 65.8 |

*Table 7.1: Test results for water simulation with isosurface generation with 262,000 Marching Cubes voxels. Different results are gathered for particle system size and particle field radius. Results are measured as an averaged FPS over 100 frames.*

| Field Radius | 32k Particles | 64k Particles | 128k Particles |
|---|---|---|---|
| 0.0625 | 6.8 | 3.6 | 1.5 |
| 0.0313 | 30.9 | 17.8 | 10.7 |
| 0.0156 | 70.9 | 46.3 | 33.1 |

*Table 7.2: Test results for water simulation with isosurface generation using 2.1 million marching cube voxels. Different results are gathered for particle system size and particle field radius. Results are measured as an averaged FPS over 100 frames.*

the rendering time. The particle system in question is an SPH system with the optimal specifications shown earlier in Chapter 6. The scenario given is simply a calm, flat surface - this way, other influences on the performance of the system (i.e. excessive movement of fluid particles) is minimized. In addition to this information, visual representation of the results is displayed in a series of screenshots below (Figures 7.12 and 7.13). To demonstrate the differences in the smoothness of the surface, the Blinn-Phong shading model is used over the Fresnel effect.

Many conclusions can be drawn from observing these results. Although smaller particle field sizes give relatively good performance benefits in relation to both Marching Cubes grid size and particle system size, the visual rendering of these systems is not as smooth as others. As shown in both Figures 7.12 and 7.13, the lower three screenshots (referring to the small field size of 0.0156) show a much more *rough* looking surface compared to those on the upper layer (referring to a field size of 0.0313).

Adding on to the first point, the rendering for a field size of 0.0625 (relatively large) produces a very smooth surface for both Marching Cubes grid sizes, with very little change when comparing increasing system sizes (these screenshots were omitted for brevity). However, as shown in the performance tables, this large field radius is much more computationally expensive, as the large field is coupled with many more particles as well as more Marching Cubes voxels when compared with smaller sizes.

At the end of the day, real-time simulation is also important. When considering FPS, it is a necessity to define which FPS values are considered to be acceptably “real-time” and which are not. Generally, anything below 15-20 FPS is starting to be too slow and the fluid loses its realism. Therefore, combinations which produce a superior visual result are:

- 2.1 million Marching Cubes voxels, 128k particles, field size of 0.0313
- 262k Marching Cubes voxels, 128k particles, field size of 0.0625

*Figure 7.12: Rendering surfaces using different configurations. The rendering method used is a Blinn-Phong shading model specifically modified to demonstrate the differences in the smoothness of the surface and in particular the polygon makeup. This particular set of screenshots uses a Marching Cubes grid size of 2.1 million voxels. The top left screenshot shows a 32k particle system size with a field radius of 0.0313. On the top right, the system with 128k particles using the same field size. On the bottom left, a 32k system size with a 0.0156 field size, and on the bottom right a system size of 128k particles using the same field size as the bottom left. The difference in smoothness of can be clearly seen in these screenshots. Looking back at Table 7.2, although the average execution time for the top two configurations are slower, it produces a smoother surface. Visual results for configurations with 262k Marching Cubes voxels is shown in the next figure.*

This would all depend on what resources are available - if the graphics card is powerful enough, the case with 2.1 million Marching Cubes voxels would be the preferable choice as it can be rendered in real-time easily. There may even be the option, given a very powerful card, to be able to use an even higher resolution Marching Cubes grid. Additionally, this does not take into account the different scenarios that are available. For example, if the simulation is not as calm and many particles are dropping individually or are sloshing around violently in the simulation, a larger field size might not be optimal, as the larger representation of the particles would not demonstrate the turbulence as well as a smaller field radius would.

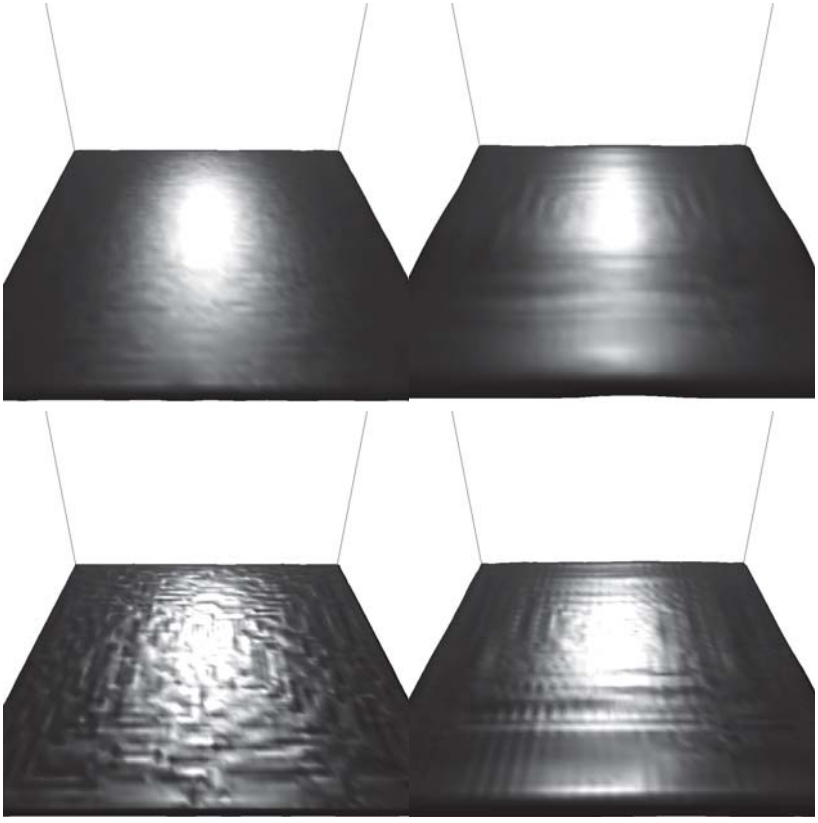

*Figure 7.13: Similar rendering results as the above screenshots, however these results are using a Marching Cubes grid size of 262,000 voxels rather than 2.1 million. At the top left, a 32k particle system size using a field radius of 0.0313. On the top right, a 128k particle system size using the same field radius. On the bottom left, a 32k system size using a 0.0156 field radius, and on the bottom right a 128k system size using the same field radius. It can be seen that reducing the Marching Cubes grid size does in general result in a poor quality surface, but using certain configurations can provide a surface which is still reasonably smooth.*

*Figure 7.14: Rendering results for a field radius of 0.0625. These are compared separately as there is not much change in varying different system sizes here. The left image represents a 64k system size with 262k Marching Cubes voxels, while the right represents a 64k system size with 2.1 million Marching Cubes voxels. These configurations produce a very smooth surface (in relation to smaller field sizes) but as shown in Tables 7.1 and 7.2, run relatively slowly.*

*Figure 7.15: Comparison between simulated fire and real fire for a standard fire scenario. It demonstrates the suitability of the method for the behavioural aspect of the simulation, where it clearly shows the tongue-licking style of effect shown in the photo [137].*

## 7.8 Realism Comparisons

When analyzing which model is preferable over another, one of the main strengths or weaknesses of a model is how well it represents the real-world phenomena it is trying to simulate - i.e. the level of *realism* it provides. However, for the purposes of comparing different approaches to come to a conclusion on which method is visually superior, realism can be intrinsically difficult to measure - it is a subjective concept, what might look realistic to one person might not be perceived the same way by another person.

In order to test this, a visual comparison has been made between the results of the simulation and a real photograph of each of the phenomena. The comparison is made using the best model choice for the photo. This comparison will attempt to show how certain features or behaviours observed in the real-world photograph are replicated in their respective simulations. All photos are in the public domain and sources are referenced.

Figure 7.15 shows a comparison between the fire simulation and a real-life photo of a large fire. The simulation has been set up with the intended behaviour seen in the photo in mind, with an emphasis on creating some turbulent behaviour for flame tongues, rather than a

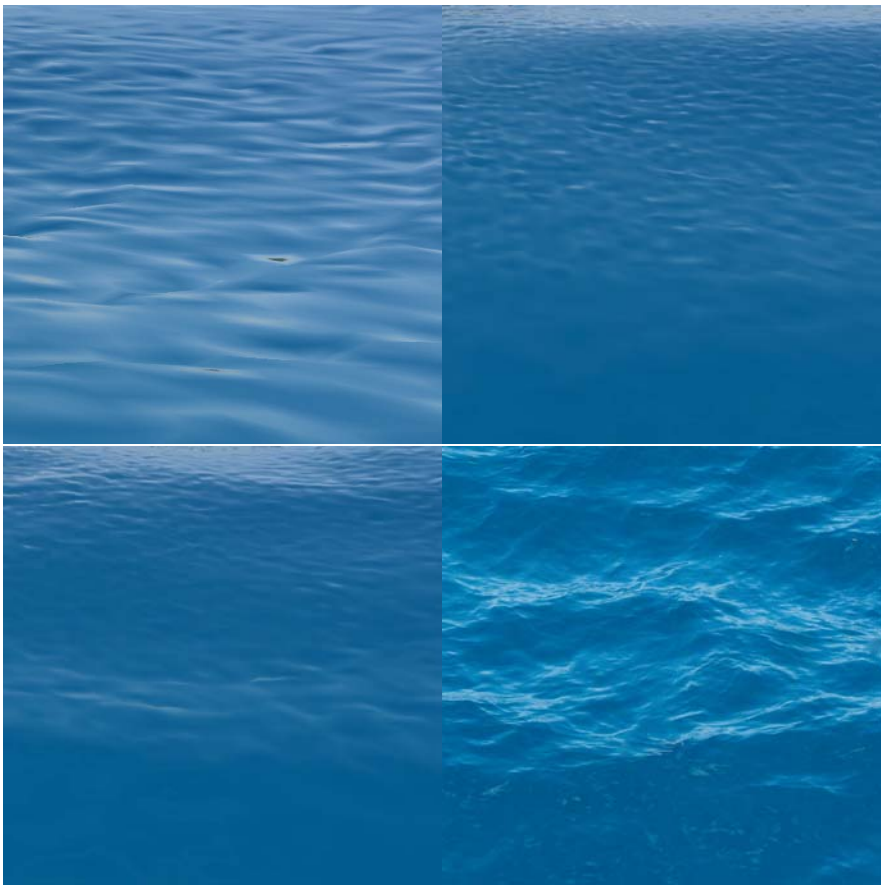

*Figure 7.16: Comparison of small waves scenario. To get this effect from the simulation, a portion of the particles have been raised and dropped out of camera view and the slight ripple effect has been observed. The comparison shows good representative behaviour of the fluid as well as some realistic looking reflections [138].*

steady upwards flow such as what would be observed from a single match for instance. These screenshots are particularly good at showing structure within the flame, where dense areas of the flame are rendered brighter. It is implied that these brighter areas are also an indication of heat levels in the system. This is consistent with the real-life comparison, as hotter areas of the flame (such as the center) are brighter yellow colours while colder areas of the flame are darker orange. These screenshots also show good features such as flame curls generated by the vortices in the flame - this is shown particularly well in the top-right screenshot.

The comparison does reveal the restrictions of the point-based method however; the colouring is good where the brighter parts of the flame are lighter, but as the particles become less dense and turn red the individual particles are more clearly seen - this is in contrast to the real photo where the fire has a continuous area of flame even at the edges. There is also a point regarding the setup of the system in general - specifically the sphere surface on which particles are emitted. This setup works in the case of using the sphere only for the purpose of inducing vorticity, however the real-life photo does not employ any sort of spherical burning source, so the overall shape of the flame would be expected to be different in this respect.

*Figure 7.17: Comparison for water droplet scenario. This scenario demonstrates the behavioural aspect fairly well but shows the limitations of the rendering. The particles are relatively large compared to what one would expect from the photo: The simulation would need a lot more particles of a smaller size and smaller field sizes to get the more defined upwards splash shape shown in the photo [136]. Nevertheless the similar splash behaviour and ripple effect is shown in the simulation.*

Figure 7.16 shows a comparison between a calm water simulation and a photo of a large body of water. Here the behaviour has been set up to imitate the calm wave-like behaviour of this photo, which seems to be of a sea or large lake. The comparison also shows how well the rendering of the simulation resembles the reflection features found in the real-life photo.

Figure 7.17 shows a comparison for a different behaviour of water, where it it compared to a photo of a droplet of water splashing. It was chosen to demonstrate a more turbulent behaviour than in Figure 7.16. The effect was created by dropping a small sphere of particles (resembling a real-life droplet) directly into the body of the particle fluid. The resulting behaviour resembles the real-life photo relatively well although the splash does not create as many ripple effects. One additional thing to note about this comparison is that the shader does not use dynamic reflection mapping - in the photo the reflection of the droplet is clearly visible in the water, while the simulation is unable to do this using the current shaders. This would be an improvement which could easily feature in future work in this area.

Figure 7.18 also demonstrates a similar effect but with a larger splash. This particular photo was difficult to replicate because of how and where the photo appears to have been taken. The water is fairly shallow and it seems like it has been taken indoors, so the general method of using a skybox to determine fragment colours needed to be modified. A flat grey colour is used when determining the refraction colour, as this represents the silver or grey object shown in the photo. If the photo was taken indoors, it can also be assumed that bright artificial light is hitting the surface from many directions, so using a very bright skybox (even a pure white one) works well in place of a standard blue sky skybox or a cloudy one. Similar to the droplet scenario, this screenshot also highlights possible improvements to this rendering method. There is noticeable colour differences on the back edge of the splash wall (furthest from the camera). In the photo,

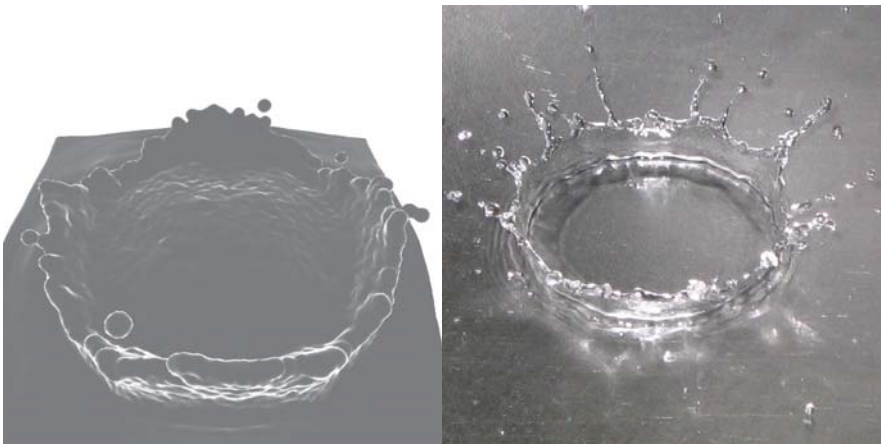

*Figure 7.18: Comparison for water splash scenario. Again, this scenario demonstrates some good behavioural aspects but also the restrictive aspect of the rendering. The splash effect is unable to get the well defined surface shown in the photo because the resolution of the mesh is too low (relative to the real-life photo). However the splash behaviour is demonstrated very well here [139].*

this brightness is actually due to light passing through the thin water layers multiple times, which is difficult to do in a simulation without using some type of ray-tracing method. The method employed here in the simulation is better suited to ocean-like scenarios, where the water is very deep and refracted light rays would not be expected to pass through more than one layer.

## 7.9 Summary

This chapter focuses on the different rendering methods implemented for the simulations of fire and water. Basic point-based sprites are used for the non-interacting particle method, while density-dependent point sprites are used for the velocity-vortex method. A point-based volumetric approach is implemented for water, but as shown in the example results, the volumetric rendering does not sufficiently render the fluid to be particularly realistic. Instead, an isosurface mesh is generated separately based on the positions of the particles in the fluid. Generating surfaces requires much more processing power and therefore the maximum number of particles that can otherwise be rendered using less computationally intensive rendering methods (such as point-based methods) is reduced. This is important when considering real-time simulation - less particles means less realistic behaviour. The trade-offs between behavioural realism and visual realism are explored and as discussed in Section 7.7, some compromises can be made between computational restrictions and visual restrictions, to achieve a realistic-looking surface rendering at an interactive rate.

# CHAPTER 8

# DISCUSSIONS AND CONCLUSIONS

## 8.1 Discussion

This thesis explored the implementation of particle systems for real-time simulation, when applied to real-world complex phenomena. In particular the phenomena of fire and water were simulated, taking into account the different behaviours that govern these systems. To analyze effectiveness of particle systems representing these phenomena in real-time, investigations over the particle model used, spatial partitioning, integration methods and rendering approaches were performed. Each phenomena was modeled using two different particle system approaches, and the strengths and weaknesses for using each one were described. Spatial partitioning and integration methods were applied generally over all systems, analyzing the strengths and weaknesses of different integration methods in general, and utilizing partitioning methods to speed up the interaction phases of the particle methods.

NVIDIA CUDA was used to parallelize these simulations, as without parallelization these approaches would be unable to represent the natural phenomena in real-time - that is to say, the system must be completely interactive at any point during which the simulation is running. In relation to video games especially, real-time rendering is a key aspect to increase realism, because the user expects to interact with the game constantly. Movies, on the other hand, do not have to enable interaction with the user. Real-time simulation of these more complex phenomena is generally not explored as much as non-real-time simulation. These post-processing methods of simulation produce more accurate and realistic representations by implementing more complex models that require significant computation per frame. Real-time implementations cannot afford this luxury, as the computation per frame must remain at an interactive rate. As demonstrated in this thesis, parallelization helps solve this problem.

Several significant aspects of real-time rendering of fire and water systems were discussed in this thesis. The following sections will provide a short summary on the findings of these aspects.

### 8.1.1 Numerical Integration and Spatial Partitioning

Integration methods are central to these simulations, both in real-time and non-real-time cases. When choosing an integration method, two important factors are the order of the method and the accuracy of the value estimation. Chapter 5 deals with choosing which methods to use and why. For non-real-time rendering, the order of the integration method is not particularly important. This is because, since it is not being simulated in real-time, there is no reason not to use a highly accurate integration method of a very high order. High-order integration methods require many passes of each step in a single simulation iteration and this allows the error introduced to be minimized. Real-time simulations cannot afford this luxury, as too many passes per iteration would cause the simulation to run slowly. It was found in Chapter 5 that energy conservation in the system is important and is conserved best over long periods of time by using *symplectic* integration methods. The global error of these symplectic methods is not necessarily minimized, but rather is almost guaranteed to maintain a constant oscillation around a certain initial energy value as long as the timestep remains constant. In contrast, non-symplectic methods such as the Euler and Runge-Kutta methods may, over time, become unstable without sufficient damping to prevent energy from increasing due to error. Taking this information into account, the Leapfrog method for integration is preferred for the particle systems as it is a symplectic method with a single iteration pass, while maintaining an error of order two (as the velocity is evaluated twice per iteration).

The implementation of a spatial grid when parallelizing particle systems is crucial to allow the system to process interactions quickly and efficiently [127]. Several different implementations of the spatial grid were explored in this thesis. One of the main issues to consider was the size of the spatial grid cells in relation to how the particles interact with each other. In the case of particles that interact only at their own physical border (for example the spring-mass model) the spatial grid cell size can be minimized. But in the case where there are forces acting over long distances between particles, the cell size must incorporate these forces and use an *influence radius* which ultimately results in a slower simulation as relatively more particles (and thus interactions to process per iteration) will exist within any one field.

In the implementation of the velocity-vortex method discussed in Chapter 6, the spatial grid is dynamic, increasing and decreasing as the particle system expands and contracts. This is important in relation to the nature of the system - the water system, for instance, is already contained within a rigid container (technically a rigid-body) so the particle system will not expand beyond a certain boundary. Fire particles, on the other hand, have no specific boundaries and can move unimpeded. Additionally, scaling the spatial grid dynamically allows the spatial grid data to be re-used when calculating the Jacobian matrix for velocity during the vorticity-from-velocity update phase - vorticity should curl away from the outer edges of the flame to ensure it does not simply flow outwards indefinitely.

An interesting option to explore would be to make the spatial grid for the water simulation scalable at least in the y-direction, because a common state for water to be in is undisturbed at the bottom end of a container. In this situation, many spatial grid cells in the upper area of the

simulation will be unused for a large period of time. Even if the system is fairly turbulent, it is rare that particles are launched out of the fluid so high. Therefore, reducing the dimension of spatial grid cells in the y-direction might be beneficial to the computation time of the system as it will free up threads which would otherwise be pre-occupied dealing with empty spatial grid cells. There are other implementations of spatial grids which have not been explored in this thesis [45], which have the possibility to provide some improvements. Harada et al. [45] in particular uses a dynamic spatial grid approach (similar to the grid used by the velocity-vortex method) which divides fluid into regions with their own bounding boxes allowing it to effectively remove the fixed grid restriction for a fluid simulation.

### 8.1.2 Choices in Fire and Water Simulation Models

Particle systems simulating the fire or flame phenomena were implemented using a non-interacting particle approach and a velocity-vortex approach. As shown in the performance graphs in Section 6.4.6, the non-interacting method outperforms the velocity-vortex method in terms of computation time, however as shown in the visual comparisons in Section 7.3, the velocity-vortex method provided a better visual representation than the non-interacting method. It was found that this superior visual representation was in large part due to the velocity-vortex method being able to provide more fluid-like behaviour of a flame (due to the underlying fluid-dynamics algorithms used by the method as described in Section 4.4) while being able to maintain interactive frame rates. Many fire simulations use Eulerian [134] [56] rather than particle-based (Lagrangian) methods. It was important to investigate particle-based methods here and it was found that velocity-vortex methods incorporate many elements of Lagrangian methods, while also using elements of the Eulerian velocity grid. Using this implementation for the fire provides the interactivity and freedom of movement given by the Lagrangian aspect of the method, while the Eulerian aspect provides the more accurate fluid dynamics required for a realistically behaving fire. As for other fire-related features, there has been a lot of interesting work in modelling the way fire *spreads* and *burns* [82] [69] which is a completely different set of methods to those used in this thesis, but is nevertheless important, especially considering the use of fire simulations in real-time media. Not only should the fire appear realistic, it should also spread to other areas and/or burn logs or other fuel sources realistically.

Although it was found that the velocity-vortex method does provide a better solution to fire simulation, the non-interacting representation does still offer a number of advantageous aspects. Firstly, the fast execution time allows the system to be represented by a much larger system size than the velocity-vortex model when being run in real-time. With no interactions to process, the system can be scaled easily until all resources on the graphics card are fully utilized. The velocity-vortex method can also scale well, but not to the same extent. This brings up the second point - poor stability can occur when using the velocity-vortex method if the system is not properly scaled and does not take into account the additional error which is ultimately generated from increasing the number of particles. In particular, interaction with the vorticity-inducing rigid-body (used to set up a flame-like shape) becomes increasingly dan-

gerous to the stability of the system as this (alongside the integration method) introduces error into the system when pushing particles back outside the body. The more particles simulated, the greater the error is likely to be and therefore the more likely the system is of becoming unstable. A way to solve this would be to implement a more accurate particle-to-rigid-body interaction algorithm, although the extent that this could affect the overall performance of the simulation is unknown.

Fire particle systems were paired with point-based rendering methods, as flames do not have a defined free surface. Combining point-based approaches with the additional data, such as fluid density per particle provided by the velocity-vortex method allowed the visual representation of the flame to be more realistic than that of the non-interacting method implementation.

Systems simulating the water phenomena were implemented using a modified spring-mass approach and a Smooth Particle Hydrodynamics (SPH) approach. As shown in Section 6.5.3 the spring-mass model approach was found to perform faster than the SPH approach only for smaller system sizes - approximately twice as fast at a system size of 16,000 particles, leveling out to about equal at 128,000 particles, after which the SPH approach began to outperform the spring-mass model. Also discussed in Section 6.5.3, the decline in the modified spring-mass model's usability was due to the increased chance of instability when the model is used for larger system sizes. While SPH methods use viscous forces to internally dampen the system and keep particles together in the fluid, the spring-mass approach does not apply contractive spring forces to pull escaping particles back into the main fluid mass, meaning that the method is more prone to instability as the number of particles in the system increases. In terms of visual results, the SPH model was able to simulate turbulent behaviour very well, while the spring-mass model was restricted to calmer behaviour scenarios. When combined with surface rendering methods and Fresnel reflection and refraction effects, these methods were shown to provide good visual representations of water in a variety of behavioural scenarios.

The spring-mass particle model was the first water particle model investigated in this thesis, building on the work of Green [41], creating a much larger system which exhibits certain fluid-like behaviour at least at the macro level. One of the suggestions of Green [41] was that the system could be improved and applied to a Smoothed Particle Hydrodynamics [86] system for greater fluid dynamics effects. SPH indeed produces a much more realistic flow movement in the particle system (as demonstrated in Figure 6.9) and is generally preferable over the spring-mass model. It is interesting to observe the changes in the fluid when altering the different parameters of the system, such as the fluid rest density, viscosity and smoothing kernels. Having a more viscous fluid, for instance, causes the fluid to stay together more strongly, resulting in a sort of rubbery or bouncy effect. Thus SPH implementations are much more flexible to different fluid configurations, for example milk or oil. SPH methods have already been used to model interactions between different types of fluids [75], though not in real-time. Other more complicated SPH methods have been implemented [57] [22] recently which improve upon the standard SPH implementation by handling larger timesteps and large-scale simulations more efficiently, although at the moment the scales described in these implementations are too large to be simulated in real-time using the current state-of-the-art hardware.

### 8.1.3 Surface Generation and Rendering

Generation of isosurfaces around or over an existing particle system is an important part of the work in this thesis, because it binds both good rendering methods and good computational methods together, as well as parallelization using CUDA. Even if the computation method is fast and efficient, it means nothing in the sense of real-time simulations if the rendering method is not adequately fast enough.

Water systems require a free surface to be generated over the existing particle system in order to appear realistic. It was shown in Chapter 7 that the Marching Cubes algorithm is able to be parallelized effectively on the architecture of modern graphics processing units to work in tandem with the underlying particle system to render a surface at interactive rates. Marching Cubes provides a good trade-off between realistic surface generation and execution at real-time speeds. Configurations when using Marching Cubes to generate the surfaces allow for customization of the method depending on whether greater surface quality or faster execution time is favoured. Faster execution time requires a lower resolution of the Marching Cubes grid, as well as smaller particle system sizes and smaller particle field sizes. Greater surface quality requires a high resolution Marching Cubes grid and large particle systems and field sizes. Optimal configurations when using the GeForce GTX 780 graphics card are provided in Section 7.7. The surface generation method was combined with Fresnel reflection and refraction mapping to the surrounding skybox environment to provide realistic visual approximations to real-world water effects.

Marching Cubes as a model for isosurface generation is fairly commonly used in computer graphics [73] but rarely is it used as a real-time approach. The method is difficult to implement in real-time when trying to get very high-quality surfaces. For example, the work in this thesis deals with Marching Cubes grids with a maximum number of voxels of 16 million (although typically only up to 2.1 million in most scenarios). Non-real-time implementations can afford to increase this number much further, making the surface generated a lot smoother. It is essential to achieve a balance of the different aspects of the surface generation, including the particle system size and the influence radius (as discussed in Section 8.1.1), which allows reasonably good surfaces to be rendered in real-time. This work builds upon that described in [112] with larger numbers of particles, together with the combination of the isosurface generation method and a proper fluid dynamics particle system.

Surface generation is specifically applied to water simulations as water has a defined surface while a flame has a less well defined boundary. Point-based sprite rendering methods are therefore better suited for fire simulations. That being said, methods incorporating point-based approaches have also been used to render water [5]. These are particularly effective for rendering water foam, for the same reasons they are effective for fire. Although this method of rendering produces a good effect, without the surface it is unable to implement effects such as Fresnel reflection, which is a far better representation of the real-world phenomena. One possibility could be to combine these two methods, with a surface generation algorithm used to generate smooth fluid portions of the water, and a point-based approach rendered on top of

the surface handling splashes and foam effects.

Other approaches to real-time rendering include the surface-particle rendering approach [132], whereby a layer of particles is laid over the top of the surface and move along with it (following SPH density and pressure rules). The surface particles are used as the vertices that make up the isosurface. In this thesis, Marching Cubes was chosen over this method because of the problem of handling particles disconnected from the fluid [132]. It is not guaranteed that fluid particles that are separated from the overall fluid - for example a droplet of water resulting from a splash - would be in proximity to enough (or even any) surface particles. If this was the case, the resulting droplet would be poorly rendered or even completely ignored by the surface generation algorithm.

## 8.2 Conclusions

The work in this thesis has shown that particle systems are able to simulate the real-world phenomena of fire and water in real-time. This is accomplished through the use of parallel implementation of the particle models equations on the GPU using CUDA and OpenGL rendering approaches specific to fire and water situations, for example surface rendering. It is shown that by providing an adequate balance between the particle system's complexity and the rendering implementation, effective results can be achieved. An in depth analysis into the use of these particle systems has also been performed, including covering different integration methods for physical accuracy and spatial partitioning methods for computational speed-up.

Research question one asked whether particle systems are able to be used effectively to simulate the behaviour of fire and water effects in real-time. This thesis described the implementation of four different types of particle models - the non-interacting model and the velocity-vortex model for fire simulation, and the modified spring-mass and SPH models for water simulation. The representation of these fire and water effects using particle systems allowed a wide variety of behaviour to be simulated by allowing individual particles to move around creating complex shapes such as droplets, waves, and flame tongues. Individual particle parameters such as viscosity, density and vorticity allow particles to employ fluid dynamics equations to move with fluid-like motion, while spring-mass motion using colliding spring forces is also able to represent fluid-like motion relatively well with large system sizes. It was also shown that using particles enabled the system to be interactive with user-manipulated rigid bodies and forces, an important feature when considering realistic simulation and immersion.

The second research question asked for an analysis on the strengths and weaknesses for using different particle models to simulate these effects. It was found that the velocity-vortex model does provide a superior visual representation, as it is able to produce swirling vortex effects and generally more fluid-like movement present in real fire phenomena, due to the underlying fluid dynamics algorithms used by the method. On the other hand, the non-interacting particle model was more reliant on randomization to demonstrate chaotic effects and was not able to reproduce this behaviour. For water simulation, the SPH model was shown as superior to the spring-mass model, as it was also able to achieve water flow behaviour which the

spring-mass model could not - in particular, turbulent behaviour (such as waves and splashes) was much better suited to the SPH model. When considering real-time interactive speeds, the size of each system was one of the main limiting factors to the method's implementation. Due to its simplicity, the non-interacting particle model was shown to be able to handle a larger amount of particles than the velocity-vortex model, but this did not make up for the lack of flame-like behaviour demonstrated by the model. The spring-mass and SPH models were able to handle similar amounts of particles while maintaining an interactive frame-rate, although relatively larger SPH systems were more stable than their spring-mass counterparts because of the underlying viscous forces present.

Integration methods of particle systems in general were also explored. The results showed that a Leapfrog integration method or similar is the most preferred approach for these types of particle systems due to the need for low-order integration methods for real-time simulation purposes. Higher-order methods are more accurate, but are slower; the Runge-Kutta 2nd and 4th order methods are two and four times slower respectively than the Leapfrog method. Because the Leapfrog method is also a symplectic method, it was able to conserve energy over a long period of time.

The third research question of this thesis asked how these particle systems can be rendered visually at real-time execution speeds. The fire particle systems used a simple point-based approach to render the effects, where point sprites were rendered brighter in areas where particles were more dense. While a point-based approach was sufficient to demonstrate flame-like behaviour, some aspects of realistic fire were represented but the method also had its weaknesses. In particular, the outer edges of the flame were frequently rendered more as points than as a continuous smooth area of flame, due to the very low density of particles in these areas. In this situation the method was better suited to rendering spark-like behaviour at the edges of the flame. The point-based approach was also able to visualize structure within the flame, such as the individual vortices making up flame curls, and the variation in temperature of the flame in areas where particles clustered in density.

The water systems utilized a surface generation approach to rendering, as water has a defined free surface when compared to a more gaseous substance like fire. Marching Cubes was chosen as the method for surface generation due to the algorithm lending itself well to parallelization and its versatility in generating surfaces over a wide variety of underlying structural shapes made by the particles. The realism of the generated surface was enhanced through the use of Fresnel shading, rendering the colours of a surrounding skybox onto the surface with realistic reflection and refraction factors. Both the fire and water rendering implementations were able to be rendered in real-time, although the surface required significantly more time to generate than simply rendering the particles as point sprites.

Research question four asked how parallelization could be used to help speed up the execution of these particle models and rendering methods.Parallelization of all particle systems described in this thesis was implemented using NVIDIA's CUDA. Due to the massively parallel architecture, the complex methods governing the behaviour of the particles for fire and water systems are able to be computed at interactive rates, while using the interoperability of CUDA

and OpenGL further increases this performance by not having to copy rendering data between the host memory and graphics card. Parallelization of the particle methods allowed systems of a much larger size to be simulated while maintaining interactivity. Parallelization of the surface generation approach achieved similar computational advantages, providing support for high resolution Marching Cubes grids (up to 16 million cells on the GeForce GTX 780) while maintaining real-time interactivity. Performance was also improved by implementing a spatial partitioning algorithm in parallel, reducing the search space of particles when processing interactions between them.

The final research question asked for suggestions when balancing computational aspects and rendering aspects of the simulation, specifically in relation to real-time rendering. Section 7.7 described in detail the complexities of combining both the particle system algorithms and the surface generation algorithm. It was found that the particle system size, integration methods, Marching Cubes grid resolution and the particle field radius were primary concerns when balancing this simulation. It was discussed that in order to remain running in real-time, combinations of these primary parameters would need to fit within the real-time simulation constraints. For example, using a higher order integration method would increase the physical accuracy of the system, but to remain interactive the surface grid resolution may need to be reduced to account for the extra time needed to execute the higher-order integration method. The combination of these aspects of this thesis demonstrated the inherent challenges of real-time rendering when applied to natural phenomena, and it was shown that in balancing these aspects it is possible to achieve some visually convincing results at an interactive frame rate.

In conclusion, the thesis produced the following contributions:

- A demonstration which shows that particles are capable of simulating fire and water realistically and at real-time speeds.
- An implementation of these particle systems in parallel with NVIDIA CUDA, providing computational speed-ups in order to maintain interactivity while allowing the system to improve in size and physical accuracy.
- A discussion on the advantages and disadvantages of using different particle models and different integration methods when simulating fire and water.
- A combination of the underlying particle system and the Marching Cubes surface generation algorithm to produce a quality rendering of water effects with real-time interactivity.
- A discussion on the trade-offs considered when balancing computational and rendering aspects of the system.

## 8.3 Future Work

Future work in this area could explore a number of different aspects. In rendering water effects especially, implementing dynamic reflection mapping would provide additional realism to the

surface, taking into account dynamic objects in the environment as well as reflections of the water itself in more turbulent scenarios. This is particularly important in video games, where players expect to interact with the water with their characters and other objects, and this should be reflected in the surface.

Incorporating more complex implementations of SPH such as the implicit method IISPH could further improve the realism of the water behaviour. Additionally, the use of a dynamic spatial grid with water simulation is a possible area for improvement computationally and could allow the particle system to maintain real-time interactive rates with larger numbers of particles.

Exploring the use of SPH for other fluids, instead of just water, is also an interesting option. Fluids such as mud or slime would have different sets of rules governing the behaviour of the fluid when compared to water. Building on this idea, Fresnel reflection and refraction rendering effects would not be suitable for simulating slime or mud effects, so additional rendering methods for these would need to be considered. Combining different types of fluid together is also a compelling option to consider. For example, water could be poured over mud, or a "rainy day" scenario could be simulated where water gathers in certain areas over time, causing slips or floods in mud-like materials. Interaction between two or more different fluids has been explored before, but doing so in real-time would provide some additional challenges. Similar to the interaction of multiple fluids, the interaction between fluids and more complex rigid-bodies could be explored, such as a boat floating on top of the water surface. This would be a perfect instance to demonstrate dynamic reflection mapping - rendering the reflection of the boat in the water.

Fire and flame simulation has a number of possible options - incorporating dual fire and smoke systems, for instance, would be a definite possibility as smoke particle systems share similar behavioural constraints as velocity-vortex methods. Additionally, interaction with environmental objects could be explored, such as how a burning object would disintegrate over time and how this would affect the generation of the flame particles and the movement of the flame.

# BIBLIOGRAPHY


[1] T. Alkenine-Mller, E. Haines, and N. Hoffman, *Real-Time Rendering Third Edition*. A K Peters, 2008.

[2] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*. Clarendon Press, 1987.

[3] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[4] A. W. Appel, "An efficient program for many-body simulation," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 85–103, 1985.

[5] F. Bagar, D. Scherzer, and M. Wimmer, "A layered particle-based fluid model for real-time rendering of water," in *In Proceedings of the 21st Eurographics conference on Rendering*, 2010.

[6] M. Balci and H. Foroosh, "Real-time 3D fire simulation using a spring-mass model," in *Proceedings of the 12th International Multi-Media Modelling Conference*, 2006, pp. 108–115.

[7] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *In Proceedings of the 25th annual conference on Computer Graphics and Interactive Techniques*, 1998, pp. 43–54.

[8] P. Bartos and J. Blazek, "Hybrid computer simulation scheme for computational study of low-temperature plasma containing micrometer-sized dust particles," *IEEE Transactions on Plasma Science*, vol. 38, No 9, pp. 2407 – 2411, 2010.

[9] M. R. Bate and A. Burkert, "Resolution requirements for smoothed particle hydrodynamics calculations with self-gravity," *Monthly Notices of the Royal Astronomical Society*, vol. 288, no. 4, pp. 1060–1072, 1997.

[10] P. Beaudoin, S. Paquet, and P. Poulin, "Realistic and controllable fire simulation," in *Proceedings of Graphics Interface (GRIN'01)*, 2001, pp. 159–166.

[11] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation*. CRC Press, 2004.

[12] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *In proceedings of the 4th annual conference on computer graphics and interactive techniques*, 1977, pp. 192–198.

[13] P. Bourke, "Implicit surfaces," 1997, http://paulbourke.net/geometry/implicitsurf/.

[14] E. Bouvier, E. Cohen, and L. Najman, "From crowd simulation to airbag deployment: particle systems, a new paradigm of simulation," *Journal of Electronic Imaging*, vol. 6, no. 1, pp. 94–107, January 1997.

[15] D. E. Breen, D. H. House, and M. J. Wozny, "A particle-based model for simulating the draping behavior of woven cloth," *Textile Research*, vol. 64, no. 11, pp. 663–685, 1994.

[16] D. E. Breen, D. H. House, and M. J. Wozny, "Predicting the drape of woven cloth using interacting particles," in *In Proceedings of the SIGGRAPH Conference on Computer Graphics and Interactive Techniques*, 1994.

[17] R. Bridson, *Fluid Simulation for Computer Graphics*. CRC Press, 2015.

[18] N. Cetin, A. Burri, and K. Nagel, "A large-scale agent-based traffic microsimulation based on queue model," in *In Proceedings of the Swiss Transport Research Conference*, 2003.

[19] A. J. Chorin, "Numerical solution of the navier-stokes equations," *Mathematics of Computation*, vol. 22, pp. 745–762, 1968.

[20] A. Colagrossi and M. Landrini, "Numerical simulation of interfacial flows by smoothed particle hydrodynamics," *Journal of Computational Physics*, vol. 191, no. 2, pp. 448–475, 2003.

[21] T. H. Cormen, C. E. Lieiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[22] J. Cornelis, M. Ihmsen, A. Peer, and M. Teschner, "IISPH-FLIP for incompressible fluids," *Computer Graphics Forum*, vol. 33, no. 2, pp. 255–262, 2014.

[23] G. Cottet and P. D. Koumoutsakos, *Vortex Methods: Theory and Practice*. Cambridge University Press, 2000.

[24] M. Coulomb, "Second mmoire sur l'lectricit et le magntisme," *Histoire de l'Acadmie Royale des Sciences*, vol. 88, pp. 578–611, 1785.

[25] M. G. Coutinho, *Guide to Dynamic Simulations of Rigid Bodies and Particle Systems*. Springer London, 2013.

[26] A. Cromer, "Stable solutions using the euler approximation," *American Journal of Physics*, vol. 49, no. 5, pp. 455–459, 1981.

[27] R. A. Dalrymple and B. D. Rogers, "Numerical modeling of water waves with the sph method," *Coastal Engineering*, vol. 53, pp. 141–147, 2006.

[28] L. H. de Figueiredo, J. de Miranda Gomes, D. Terzopoulos, and L. Velho, "Physically-based methods for polygonization of implicit surfaces," in *In Proceedings of the Conference on Graphics Interface*, 1992, pp. 250–257.

[29] K. S. Delibasis, G. K. Matsopoulos, N. A. Mouravliansky, and K. S. Nikita, "A novel and efficient implementation of the marching cubes algorithm," *Computerized Medical Imaging and Graphics*, vol. 25, no. 4, pp. 343–352, 2001.

[30] M. Desbrun and M. Gascuel, "Smoothed particles: a new paradigm for animating highly deformable bodies," in *In Proceedings of the Eurographics workshop in Computer Animation and Simulation*, 1996, pp. 61–76.

[31] Y. Dobashi, Y. Matsuda, T. Yamamoto, and T. Nishita, "A fast simulation method using overlapping grids for interactions between smoke and rigid objects," *Computer Graphics*, vol. 27, no. 2, pp. 477–486, 2008.

[32] B. Eberhardt, A. Weber, and W. Strasser, "A fast, flexible particle-system model for cloth draping," *IEEE Computer Graphics and Applications*, vol. 16, pp. 52–59, 1996.

[33] D. S. Ebert, "Advanced modelling techniques for computer graphics," *ACM Computing Surveys*, vol. 28, no. 1, pp. 153–156, March 1996.

[34] R. E. English, L. Qiu, Y. Yu, and R. Fedkiw, "Chimera grids for water simulation," in *In Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013, pp. 85–94.

[35] J. P. Freidberg, *Ideal Magnetohydrodynamics*. Plenum Press, NY, 1987.

[36] A. Ghizzo, P. Bertrand, M. M. Shoucri, T. W. Johnston, E. Fijalkow, and M. R. Feix, "a vlasov code for the numerical simulation of stimulated raman scattering," *Journal of Computational Physics*, vol. 90, no. 2, pp. 431–457, 1990.

[37] P. Gibbon, R. S. B. Berberich, A. Karmakar, L. Arnold, and M. Masek, "Plasma simulation with parallel kinetic particle codes," in *NIC Symposium*, 2010.

[38] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics - theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, 1977.

[39] M. E. Goss, "A real time particle system for display of ship wakes," *IEEE Computer Graphics and Applications*, vol. 10, no. 3, pp. 30–35, 1990.

[40] M. Gourlay, "Fluid simulation for video games," Online Tutorial Series - Intel Developer Zone, 2012.

[41] S. Green, "Particle simulation using CUDA," NVIDIA Toolkit, 2012.

[42] S. Green, "Volumetric particle shadows," NVIDIA Toolkit, 2012.

[43] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, 1987.

[44] T. Harada, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. Real-Time Rigid Body Simulation on GPUs (29).

[45] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Sliced data structure for particle-based simulations on gpus," in *In Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques*, 2007, pp. 55–62.

[46] T. Harada, M. Tanaka, S. Koshizuka, and Y. Kawaguchi, "Real-time coupling of fluids and rigid bodies," in *In Proceedings of APCOM '07 in conjunction with EPMESC XI*, 2007.

[47] M. Harris, S. Sengupta, and J. D. Owens, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. Parallel Prefix Sum (Scan) with CUDA (39), pp. 851–876.

[48] M. J. Harris, *GPU Gems*. Addison-Wesley Professional, 2004, ch. Fast fluid dynamics simulation on the GPU (38), pp. 637–665.

[49] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Interactive evolution of particle systems for computer graphics and animation," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 418–432, 2009.

[50] K. A. Hawick, "Visualizing multi-phase lattice gas fluid layering simulations," in *In Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV'11)*, 2011, pp. 3–9.

[51] K. A. Hawick and M. G. B. Johnson, "Bit-packed damaged lattice potts model simulations with CUDA and GPUs," in *In Proceedings of the International Conference on Modelling, Simulation and Identification (MSI 2011)*, 2011.

[52] K. A. Hawick, D. P. Playne, and M. G. B. Johnson, "Numerical precision and benchmarking very-high-order integration of particle dynamics on gpu accelerators," in *Proceedings of the International Conference on Computer Design (CDES'11)*, 2011, pp. 83–89.

[53] E. Hecht, *Optics, 4th Edition*. Addison-Wesley, 2002.

[54] D. Helbing, "Traffic and related self-driven many-particle systems," *Reviews of Modern Physics*, vol. 73, pp. 1067–1141, 2001.

[55] W. G. Hoover, T. G. Pierce, C. G. Hoover, J. O. Shugart, C. M. Stein, and A. L. Edwards, "Moleular dynamics, smoothed-particle applied mechanics, and irreversibility," *Computers & Mathematics with Applications*, vol. 28, no. 10-12, pp. 155–174, 1994.

[56] C. Horvath and W. Geiger, "Directable, high-resolution simulation of fire on the GPU," *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 41:1 – 41:8, 2009.

[57] M. Ihmsen, J. Cornelis, B. Solenthaler, C. Horvath, and M. Teschner, "Implicit incompressible SPH," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 426–435, 2014.

[58] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, *GPU Gems*. Addison-Wesley Professional, 2003, ch. Volume Rendering Techniques (39).

[59] M. G. B. Johnson, D. P. Playne, and K. A. Hawick, "Data-parallelism and gpus for lattice gas fluid simulations," in *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, 2010.

[60] J. T. Kajiya and T. L. Kay, "Rendering fur with three dimensional textures," *Computer Graphics*, vol. 23, no. 3, pp. 271–280, 1989.

[61] M. Kass and G. Miller, "Rapid, stable fluid dynamics for computer graphics," *Computer Graphics*, vol. 24, no. 4, pp. 49–57, 1990.

[62] A. E. Kaufman, "Voxels as a computational representation of geometry," in *In the Computational Representation of Geometry SIGGRAPH '94 Course Notess*, 1994.

[63] A. J. Kemp, B. I. Cohen, and L. Divol, "Integrated kinetic simulation of laser-plasma interactions, fast-electron generation and transport in fast ignition," *Physics of Plasmas*, vol. 17 (5), pp. 1 – 7, 2010.

[64] P. Kipfer, M. Segal, and R. Westermann, "UberFlow: A GPU-Based Particle Engine," in *In Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2004.

[65] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., 2010.

[66] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2004, pp. 123–150.

[67] J. Krger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3D flows," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 6, pp. 744–756, 2005.

[68] J. T. Larsen and S. M. Lane, "HYADES - A plasma hydrodynamics code for dense plasma studies," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 51, no. 1-2, pp. 179–186, 1994.

[69] H. Lee, L. Kim, M. Meyer, and M. Desbrun, "Meshes on fire," in *In Proceedings of Eurographics Workshop on Computer Animation and Simulation*, 2001.

[70] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting graphical processing units for data-parallel scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, 2009.

[71] A. Leonard, "Vortex methods for flow simulation," *Journal of Computational Physics*, vol. 37, pp. 289–335, 1980.

[72] P. C. Liewer, V. K. Decyk, J. M. Dawson, and G. C. Fox, "A universal concurrent algorithm for plasma particle-in-cell simulation codes," in *In Proceedings of the third conference on Hypercube concurrent computers and applications*, vol. 2, 1989.

[73] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[74] F. Losasso, F. Gibou, and R. Fedkiw, "Simulating water and smoke with an octree data structure," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 457–462, 2004.

[75] F. Losasso, T. Shinar, A. Selle, and R. Fedkiw, "Multiple interacting liquids," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 812–819, 2006.

[76] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," *The Astronomical Journal*, vol. 82, pp. 1013–1024, 1977.

[77] T. S. Lyes, M. G. B. Johnson, and K. A. Hawick, "Visual simulation of a multi-species coloured lattice gas model," in *In Proceedings on the International Conference on Scientific Computing*, 2012.

[78] M. Macklin, M. Mller, N. Chentanez, and T. Kim, "Unified particle physics for real-time applications," *ACM Transactions on Graphics*, vol. 33, no. 4, p. Article 153, 2014.

[79] S. Markidis, G. Lapenta, and Rizwan-uddin, "Multi-scale simulations of plasma with iPIC3D," *Mathematics and Computers in Simulation*, vol. 80, pp. 1509 – 1519, 2010.

[80] N. S. Martys and R. D. Mountain, "Velocity verlet algorithm for dissipative-particle-dynamics-based models of suspensions," *Physical Review E*, vol. 59, no. 3, pp. 3733–3736, 1999.

[81] R. I. McLachlan and P. Atela, "The accuracy of symplectic integrators," *Nonlinearity*, vol. 5, no. 2, p. 541, 1992.

[82] Z. Melek and J. Keyser, "Interactive simulation of burning objects," in *In Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, 2003.

[83] G. Miller and A. Pearce, "Globular dynamics: A connected particle system for animating viscous fluids," *Computers & Graphics*, vol. 13, no. 3, pp. 305–309, 1989.

[84] J. J. Monaghan, "An introduction to SPH," *Computer Physics Communications*, vol. 48, pp. 89–96, 1988.

[85] J. J. Monaghan, "Smoothed particle hydrodynamics," *Annual Review of Astronomy and Astrophysics*, vol. 30, pp. 543–574, 1992.

[86] J. J. Monaghan, "Smoothed particle hydrodynamics," *Reports on Progress in Physics*, vol. 68, pp. 1703–1759, 2005.

[87] A. Mller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003, pp. 154–159.

[88] C. D. Norton, B. K. Szymanski, and V. K. Decyk, "Object oriented parallel computation for plasma simulation," *Communications of the ACM - Special issue on object-oriented experiences and future trends*, vol. 38, no. 10, pp. 88–100, 1995.

[89] E. Novikov, "Generalized dynamics of three-dimensional vortical singularities (vortons)," *Zh Eksp Teor Fiz*, vol. 84, p. 981, 1983.

[90] *CURAND Library Programming Guide*, NVIDIA.

[91] *NVIDIA CUDA C Programming Guide Version 5.5*, NVIDIA.

[92] NVIDIA, *Thrust Quick Start Guide*.

[93] NVIDIA, *Tuning CUDA Applications for Kepler*.

[94] NVIDIA, *Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Fermi*.

[95] NVIDIA, *Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*.

[96] L. Nyland, M. Harris, and J. Prins, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. Fast N-Body Simulation with CUDA (31), pp. 677–695.

[97] R. C. S. Oliveira, C. Esperana, and A. Oliveira, "Exploiting space and time coherence in grid-based sorting," in *In Proceedings of the 2013 26th SIBGRAPI - Conference on Graphics, Patterns and Images*, 2013.

[98] opengameart.org, "Opengameart website for open-source high quality images for use in computer graphics programs and games."

[99] W. Park, E. V. Belova, G. Y. Fu, X. Z. Tang, H. R. Strauss, and L. E. Sugiyama, "Plasma simulation studies using multilevel physics models," *Physics of Plasmas*, vol. 6 (5), pp. 1796 – 1803, 1999.

[100] A. G. Peeters and D. Strintzi, "The fokker-planck equation, and its application in plasma physics," *Annalen der Physik*, vol. 17, no. 2-3, pp. 142–157, 2008.

[101] M. Pharr and G. Humphreys, *Physically-Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2010.

[102] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, 1975.

[103] E. C. Pickering, "Applications of fresnel's formula for the reflection of light," in *In Proceedings of the American Academy of Arts and Sciences*, vol. 9, 1873.

[104] M. Planck, *The Theory of Heat Radiation*. P. Blakinston Son & Co, 1914.

[105] D. P. Playne and K. A. Hawick, "Classical mechanical hard-core particles simulated in a rigid enclosure using multi-gpu systems," in *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2012.

[106] D. P. Playne, K. Hawick, and M. G. B. Johnson, "Simulating and benchmarking the shallow-water fluid dynamical equations on multiple graphical processing units," in *In Proceedings of the Twelfth Australasian Symposium on Parallel and Distributed Computing*, 2014.

[107] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007.

[108] X. Provot, "Deformation constraints in a mass-spring model to describe rigid cloth behavior," in *Graphics Interface*, 1996, pp. 147 – 154.

[109] H. Ray, H. Pfister, D. Silver, and T. A. Cook, "Ray casting architectures for volume visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 3, pp. 210–223, 1999.

[110] W. T. Reeves, "Particle systems – a technique for modeling a class of fuzzy objects," *ACM Transactions on Graphics*, vol. 17, no. 3, pp. 359–376, 1983.

[111] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics and Interactive Techniques*, 1987, pp. 25–34.

[112] I. D. Rosenberg and K. Birdwell, "Real-time particle isosurface extraction," in *In Proceedings of the 2008 Symposium on Interactive 3D graphics and games*, 2008, pp. 35–43.

[113] J. Rychlewski, "On hooke's law," *Journal of Applied Mathematics and Mechanics*, vol. 48, no. 3, pp. 303–314, 1984.

[114] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[115] C. Schlick, "An inexpensive BRDF model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994.

[116] R. Scott, "Sparking life: notes on the performance capture sessions for the lord of the rings: the two towers," in *In Proceedings of ACM SIGGRAPH Computer Graphics*, 2003.

[117] A. Selle, M. Lentine, and R. Fedkiw, "A mass spring model for hair simulation," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 64::1–11, 2008.

[118] K. Sims, "Particle animation and rendering using data parallel computation," *Computer Graphics*, vol. 24, no. 4, pp. 405–413, 1990.

[119] J. Stam, "Stable fluids," in *In Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999.

[120] J. Stam and E. Fiume, "Depicting fire and other gaseous phenomena using diffusion processes," in *In Proceedings of the 22nd annual conference on Computer Graphics and interactive techniques*, 1995.

[121] D. Stora, P. Agliati, M. Cani, F. Neyret, and J. Gascuel, "Animating lava flows," in *In Proceedings of the 1999 conference on Graphics Interface*, 1999.

[122] T. Sugiyama and K. Kusano, "Multi-scale plasma simulation by the interlocking of magnetohydrodynamic model and particle-in-cell kinetic models," *Journal of Computational Physics*, vol. 227, pp. 1340 – 1352, 2007.

[123] R. D. Sydora, *Advanced Methods for Space Simulations*. TERRAPUB, Tokyo, 2007, ch. Particle-in-Cell Plasma Simulation Model: Properties and Applications, pp. 47–60.

[124] R. Szeliski and D. Tonnesen, "Surface modeling with oriented particle systems," *Computer Graphics*, vol. 26, pp. 185–194, 1992.

[125] H. Takeda, S. M. Miyama, and M. Sekiya, "Numerical simulation of viscous flow by smoothed particle hydrodynamics," *Progress of Theoretical Physics*, vol. 92, no. 5, pp. 939–960, 1994.

[126] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *In Proceedings of the Vision, Modeling and Visualization Conference*, 2003.

[127] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. P. Cani, F. faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 24, no. 1, pp. 61–81, 2005.

[128] J. Tessendorf, "Simulating ocean water," 2001, part of the SIGGRAPH Course on Simulating Nature: Realisitic and Interactive Techniques.

[129] J. C. Thibault and I. Senocak, "CUDA implementation of a navier-stokes solver on multi-gpu desktop platforms for incompressible flows," in *In Proceedings of the 47th Aerospace Sciences Meeting and the New Horizions Forum and Aerospace Exposition*, 2009.

[130] F. Triquet, P. Meseure, and C. Chaillou, "Fast polygonization of implicit surfaces," in *In Proceedings of the 9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2001, pp. 283–290.

[131] K. Tsubota, S. Wada, and T. Yamaguchi, "Particle method for computer simulation of red blood cell motion in blood flows," *Computer Methods and Programs in Biomedicine*, vol. 83, pp. 139 – 146, 2006.

[132] K. van Kooten, G. van den Bergen, and A. Telea, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. Point-Based Visualization of Metaballs on a GPU (7), pp. 123–148.

[133] W. Wei and Y. Huang, "Real-time flame rendering with GPU and CUDA," *International Journal of Information Technology and Computer Science*, vol. 3, no. 1, pp. 40–46, 2011.

[134] X. Wei, W. Li, K. Mueller, and A. Kaufman, "Simulating fire with texture splats," in *In Proceedings of the IEEE Conference on Visualization*, 2002, pp. 227–234.

[135] D. Weiskopf, *GPU-Based Interactive Visualization Techniques*. Springer Science & Business Media, 2006.

[136] wikimedia.org, "Wikimedia commons - droplet picture," wikipedia/commons/c/cc/Water_drop_impact_on_a_water-surface_-_(1).jpg.

[137] wikimedia.org, "Wikimedia commons - fire brazier picture," /wikipedia/commons/b/b5/Fire_from_brazier.jpg.

[138] wikimedia.org, "Wikimedia commons - water background," /wikipedia/commons/1/1c/Blue_water_Lake_Tahoe.jpg.

[139] wikimedia.org, "Wikimedia commons - water splash," /commons/4/49/Water_splashes_001.jpg.

[140] G. S. Winckelmans and A. Leonard, "Contributions to vortex particle methods for the computation of three-dimensional incompressible unsteady flows," *Journal of Computational Physics*, vol. 109, no. 2, pp. 247–273, 1993.

[141] M. Wloka, "Fresnel reflection," nVIDIA SDK.

[142] C. Wojtan, P. J. Mucha, and G. Turk, "Keyframe control of complex particle systems using the adjoint method," in *Proceedings of the ACM SIGGRAPH Symposium on Computer Animation*, 2006, pp. 15–23.

[143] R. Yagel, "Towards real time volume rendering," in *In Proceedings of GRAPHICON'96*, 1996, pp. 230–241.

[144] F. Zhang, L. Hu, J. Wu, and X. Shen, "A SPH-based thod for interactive fluids simulation on the multi-GPU," in *In Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, 2011.

[145] Y. Zhang, C. D. Correa, and K. Ma, "Graph-based fire synthesis," in *In Proceedings of the SIGGRAPH Symposium on Computer Animation*, 2011.

[146] Y. Zhu and R. Bridson, "Animating sand as a fluid," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 965–972, 2005.