Lagrangian Liquid Simulation using SPH

Perseedoss Rajiv
MSc Computer Animation and Visual Effects
Bournemouth University, NCCA

August 19, 2011

Table Of Contents

List of Figures

Chapter 1

Introduction

Fluid simulation is a very active topic of scientific research and falls under the major field of Computational Fluid Dynamics. It provides essential insights into the workings of various systems, such as medical surgery, mechanical engineering, space travel, and etc. With computers getting cheaper and more powerful, fluid simulation gets more accurate, yet more accessible to other fields as well; visual effects is one such field where the simulation of fluid, particularly of liquids, is getting very popular both in movies, such as *Day After Tomorrow*, and games, such as *Bioshock*, where it adds a wholly new level of realism and enriches the visual experience of the audience. The simulation of natural phenomenon is an essential component of CGI movies and determines their success, such as the very realistic simulation of water in *Horton Hears A Who*.
Fluid simulation is very hard, yet very exciting as a topic of research and this provides the major motivation for the simulation of liquid to being the topic of this master thesis. Various models are used to approximate the behaviour of liquids, with the most popular one being the Navier-Stokes equation. There are basically three main approaches to simulating fluids, namely Eularian, Lagrangian and a hybrid of these two. The Eularian method uses a grid and is suited to fluids within a boundary; the Lagrangian approach contrasts totally with the latter by using particles and is best suited to interactive simulations and free surfaces (Kelager, 2006). This project involves the implementation of a particle-based Lagrangian simulation of liquid, with the Smoothed-Particle Hydrodynamics, SPH, method used to solve the Navier-Stokes equation. While a simulation consists of the simulation itself and a visualisation part, the scope of this project has been limited to the simulation mainly and OpenGL and sphere primitives are used for real-time display. Functionality has been implemented, though, to export the simulation data to an external rendering package, like Houdini, for more realistic visualisation of the simulation.
The following chapters give a summary of some related work and a brief literature review. Design considerations for the solution are then highlighted, followed by a detailed description of its implementation. This is followed by simulation considerations, scenarios and results. The report concludes with an evaluation of the implementation, from technical aspects to the implemented functionalities themselves, and critically analyses the success of the project, as well as discusses improvements and future work.

Chapter 2

Previous Work

In 1822, Claude Navier and in 1845, George Stokes formulated the Navier-Stokes Equations to model the flow of fluids (Müller et al., 2003). Smoothed-Particle Hydrodynamics (SPH) was developed by Lucy (1977) and Gingold and Monaghan (1982) for the simulation of astrophysical problems (Hoetzlein and Höllerer, 2009). In 1983, Reeves introduced particle systems for simulating fuzzy objects. Miller and Pearce (1989) introduced particle-based methods to simulate viscous liquids and melting (Becker and Teschner, 2007). SPH was first applied to free surface flows by Monaghan (1994), since it adapts very well to the simulation of complex and free surfaces. Stam and Fiume (1995) have been the first to apply SPH in the simulation of gas and fire phenomena. Following the latter were Desbrun and Cani (1996) who used SPH to animate highly deformable bodies. In 1999, Stam proposed a grid based stable semi-Lagrangian advection method that works very well for the real-time simulation of fluids. Takeshita et al. (2003) used particle methods for explosive flames. Müller et al. (2003) and Müller et al. (2005) were among the first ones to present a real-time implementation of SPH to capture the dynamic splashing effects of water, using Dynamic Air Particles. The latter also demonstrated the simulation of multiple fluids using Interface Tension Forces. They used a method of Spatial Hashing developed by Teschner et al. (2003) for optimised neighbour search. In 2005, Clavet et al. introduced viscoelastic properties to SPH.

So far, the calculation of pressure from density values has been done using the ideal gas equation; the latter was designed to work with gas, which is very compressible and gives undesired bounciness when applied to liquids (Becker and Teschner, 2007). Various works have been done to tackle the high compressibility issue, among which, Premoze et al. (2003), who introduced the Moving Particles Semi-implicit method. However, the latter needs to solve math-intensive and time-consuming Poisson equations and is therefore not suitable in real-time applications. Becker and Teschner (2007), on the other hand, simulated free surface flows using weakly compressible SPH which is faster than MPS and helps limit the density fluctuation to less than 1%. A number of work have also been undertaken to simulate non-Newtonian fluids, such as Carlson et al. (2002) who designed methods for melting and flowing, Steele et al. (2004) who represented fluid behaviour with functions and Paiva et al. (2009), who investigated on viscoplastic fluids that change viscosity with force changes.

In the recent years, the use of GPU has become very popular to process more intense computations, thus an increased number of particles, resulting in more accurate simulations and better visualisation with techniques such as isosurface ray-tracing. Harris (2003) used GPU for the simulation and rendering of fluids, more specifically of clouds. Fan et al. (2004) implemented a parallel flow simulation using the Lattice Boltzmann Model (Succi et al., 1991) on a GPU cluster and have simulated the dispersion of airborne contaminants in the Times Square area of New York City. Kolb and Cuntz (2005) presented an approach to implement a Lagrangian particle-based fluid simulation on the GPU. Harada et al. (2007) presented a GPU implementation of SPH that allowed the simulation of a massive amount of particles in real-time. Umenhoffer and Szirmay-Kalos (2008) implemented a distributed simulation, based on the Eulerian solution of the Navier-Stokes equations that runs on a GPU cluster.

The work of Müller et al. (2003), Müller et al. (2005) and Kelager (2006) form the main reference for the implementation of this project. The SIGGRAPH course notes of Bridson and Müller-Fischer (2007) gives a good description of fluid simulation. Monaghan (2005) also gives a thorough and comprehensive introduction of SPH (Becker and Teschner, 2007).

Chapter 3

Technical Background

A Lagrangian particle-based approach has been employed, together with SPH used to solve the Navier-Stokes equation and implement the liquid simulation in this project. Information about these techniques is given in the following sections. The implementation details and formulas have been skipped and would be presented, in great length, in the implementation chapter later on.

3.1 Simulation Approach

Fluids refer to either gas or liquids; in this project, it specifically refers to the latter one. A fluid is represented in a continuous space by several fields, such as pressure, velocity, density, temperature, that altogether define its behaviour (Auer, 2008).
There are two main approaches to the simulation of fluids, namely Eulerian and Lagrangian. The Eulerian method subdivides the fluid space into fixed cells and the latter model the evolution of the various fields. As such, the simulation depends both on the simulation time as well as the position of the cells. The transportation of the field quantities around the space is done through a process called advection (Bridson and Müller-Fischer, 2007).
On the other hand, the Lagrangian method uses particles to model the various fields. These particles contain all the attributes required to model a specific fields value at a specific time in the simulation. They also contain positions and move throughout the simulation space. As such, the Lagrangian method is not bound to a fixed grid and is most often used to simulate fluids in a free space.
The Eulerian method gives high visual accuracy, but suffers from mass loss and is typically slow. Lagrangian simulations, on the other hand, are faster, more memory efficient and easier to implement, thus they are better suited to interactive applications, such as video games. While the Lagrangian method inherently prevents mass loss, it cannot represent fluid surfaces quite well (Pelfrey, 2010)(Becker and Teschner, 2007).

3.2 Fluid Theory And Navier-Stokes Equation

An isothermal fluid is made up of three fields, velocity v, pressure p and density ρ (Muller et al., 2003). The evolution of these fields is governed by two equations, the conservation of mass and the conservation of momentum (Müller et al., 2003).

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0 \tag{3.1}$$

The Lagrangian method (3.1) consists of a constant number of particles, each having a fixed mass; as such, the conservation of mass is inherent and the corresponding equation can be omitted (Müller et al., 2003). The conservation of momentum is given the Navier-Stokes equation and is used to describe the movement and evolution of the fluid. The following is the Navier-Stokes equation for incompressible fluids:

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \nabla v \right) = -\nabla p + \mu \nabla^2 v + \rho g \tag{3.2}$$

where g is an external force and μ is the viscosity coefficient.

The Lagrangian particles move with the fluid and carry the fluid's attributes in space. As such, the convective derivative, $v\cdot \nabla v$, which gives the advection in the Eulerian method, can be replaced by the substantial derivative $\frac{Dv}{Dt}$ (Horvath and Illes, 2007). The Lagrangian Navier-Stokes equation therefore becomes:

$$\rho \frac{dv}{dt} = -\nabla p + \mu \nabla^2 v + \rho g \tag{3.3}$$

The Navier-Stokes equation is the application of Newton's second law of motion for fluids; as such, the three terms making up the equation compute internal and external forces on the fluid, and can be differentiated to get the acceleration a and the velocities of the particles. The mass m is replaced by the density ρ here, since the "mass in a volume", given by the density, is more applicable for fluids (Auer, 2008).

$$a = \frac{F}{m} \tag{3.4}$$

$$a = \frac{\left(-\nabla p + \mu \nabla^2 v + \rho g\right)}{\rho} \tag{3.5}$$

The three terms contribute to the internal and external force acting on the fluid, which result in the acceleration and thus movement of the fluid particles.

The first term gives the pressure force

$$-\nabla p \tag{3.6}$$

and this is what holds the fluid molecules together without it, the whole fluid collapses onto the ground. The pressure force is created whenever there is a pressure difference and it acts in the direction from high pressure to low pressure along the negative of the pressure gradient (Auer, 2008). The pressure depends on the density of the fluid, with a higher concentration of mass in a volume, or a higher density, giving rise to an increased force (from Newton's second law), and this increases the pressure (from the definition of pressure, pressure is force per area). As such, the pressure term aims to equalise the density differences throughout the fluid (Pelfrey, 2010).

The second term gives the viscosity force

$$\mu\nabla^2 v \tag{3.7}$$

and this provides the resistance to the flow of particles, hence it defines how viscous the liquid is, such as in the case of water against honey. The Laplacian of the velocity field, $\nabla^2 v$, gives the divergence of the latter from its average value and thus, this force aims to restore or smooth this velocity difference. The viscosity coefficient μ defines how strongly the force acts.

The last term

$$\rho g \tag{3.8}$$

encompasses the external forces external, such as gravity or some wind or a user-generated interaction or surface tension, and will be discussed in more details in a later section.

3.3 Smoothed-Particle Hydrodynamics

The Lagrangian particles are finite in nature and are not enough to represent the continuous field of a fluid. SPH is an interpolation method that uses these particles as discrete samples to evaluate the state of the field at any position within the fluid space; each particle is thought of as occupying a fraction of the problem space (Kelager, 2006). The interpolation is done through the use of radial symmetrical smoothing kernels, such that the approximation of a quantity A at some particle position r is given as a weighted sum of contributions from neighbouring particles (Müller et al., 2003) as follows:

$$A(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h) \tag{3.9}$$

The distance h is the core radius and determines the amount of particles that contributes to the approximation value at position $\mathbf{r}$. Beyond the distance h, other particles have no effect on the calculation.

The SPH fluid equations also require the calculation of derivatives of the quantities. These affect only the smoothing kernel function, as follows:

$$\nabla A(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r - r_j, h) \tag{3.10}$$

$$\nabla^2 A(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(r - r_j, h) \tag{3.11}$$

The pressure and viscosity forces of the Navier-Stokes equation, described in the previous section, can be easily calculated using SPH. However, these result in non symmetric forces that would introduce instabilities in the simulation. Various alternatives have been developed and this will be discussed in details in the implementation chapter later on.

Although SPH is very easy to understand and calculate, it was originally designed for compressible fluids and as such has to be carefully controlled to give visually pleasing simulation of liquids. The compressibility issue is still an active research topic for SPH.

3.4 Smoothing Kernel

The stability and accuracy of SPH calculations depend highly on the smoothing kernels. Moreover, since these calculations are done every single step of the simulation, they directly influence the speed of the simulation. Muller et al., 2003, proposed the following three kernels:

- Poly6 Kernel
  This is the fastest kernel to compute, since r appears only squared, and is used in most of the calculations (Auer, 2008).

$$W_{poly6}(\mathbf{r}, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3 & , 0 \le r \le h \\ 0 & , \text{otherwise} \end{cases} \tag{3.12}$$

$$\text{with } r = ||\mathbf{r}||$$

$$\nabla W_{poly6}(\mathbf{r}, h) = -\mathbf{r}\frac{945}{32\pi h^9}(h^2 - r^2)^2 \tag{3.13}$$

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = -\frac{945}{32\pi h^9}(h^2 - r^2)(3h^2 - 7r^2) \tag{3.14}$$

- Spiky Kernel
  This gradient of this kernel is used instead for the pressure calculation because the gradient of the Poly6 kernel goes to zero near the centre; this would cancel any repulsive force between particles that get too close to each other and thus would cause them to clump together (Auer, 2008).

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45}{\pi h^6}\frac{\mathbf{r}}{||\mathbf{r}||}(h - r)^2 \tag{3.15}$$

- Viscosity Kernel
  The Laplacian of the Poly6 kernel also give negative values that would result in high-energy viscosity forces and cause high-speed particles to accelerate instead of slowing down. The viscosity kernel is used instead; it always gives a positive Laplacian to make the viscosity force act as a damping force in all cases and help reduce the velocity of high-energy particles (Kelager, 2006).

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = \frac{45}{\pi h^6}(h - r) \tag{3.16}$$

The following figure gives a visualisation of the three kernels:

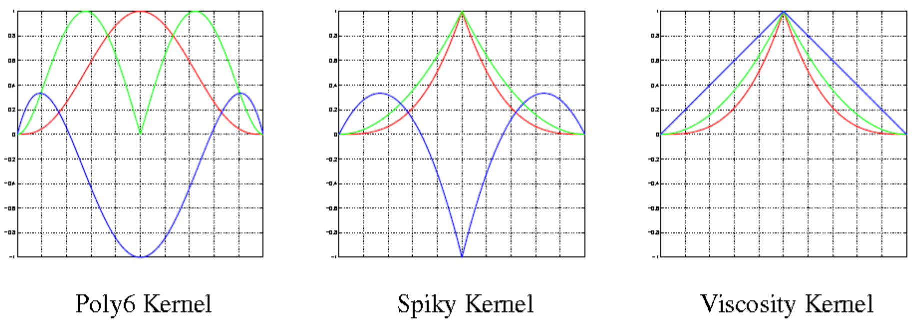


Figure 3.1: Kernels used in the SPH method. (Red) Kernel value (Green) Gradient of kernel (Blue) Laplacian of kernel. Note that the kernels and the individual curves are scaled differently in these plots. (Angst, 2007)

Chapter 4

Design

This chapter discusses all the design considerations required for the implementation of the fluid simulation. These will form the basis of the final implementation and will influence the functionalities implemented by the end of the project.

4.1 Functional Requirements/Objectives

The aim of this project is to implement a 3D Lagrangian fluid solver using SPH to simulate the flow of liquids. The simulation will be based on the forces of the Navier-Stokes equation discussed in the previous chapter, with some additional forces added to it. The following are some requirements of the solver:

- Simulation of the flow of liquids such as water, honey and etc.
- User interface to tweak the simulation parameters interactively.
- Interaction with boundary containers, such as boxes, and rigid bodies, such as spheres and mixers.
- Injection of particles, through a hose, to simulate effects such as a water jet, rain, and etc.
- Simulation of the interaction of multiple fluids.
- Real-time visualisation of fluids using particles.
- Export of simulation data to an external application, such as Houdini, for advanced rendering.

These objectives and functionalities will be adapted and updated throughout the implementation.

4.2 Entity Relationship Diagram

The following diagram shows the software architecture and the various classes that make up the solution and the relationship linking them together. These relationships are directly reflected in the implementation of the software, through class inheritance and instantiation.

The following classes form the backbone of the software:

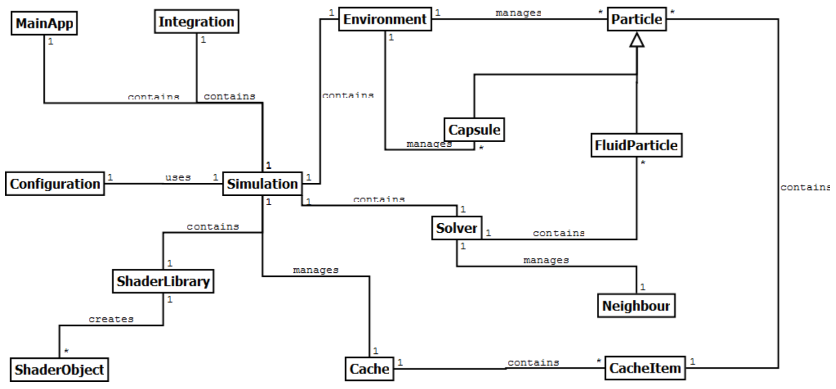


Figure 4.1: Entity Relationship Diagram, showing the global system architecture.

- Solver
  The heart of the simulation is the solver that links to all other classes and performs the fluid simulation. This class stores all the fluid particles, manage their movements through the various forces, and ensure they are confined to the boundary. It also renders the particles for visualisation.

- Simulation
  This class monitors the entire simulation, creating and managing all the other classes, including the solver. It also serves to communicate with the user interface and update the solver accordingly.

- Particle and FluidParticle
  Particles are the basic cells of a Lagrangian fluid solver. The Particle class is the parent class that stores basic attributes for a particle, such as position, velocity and etc. It is used mainly for sphere rigid bodies. On the other hand, FluidParticle is specialised and stores all the required attributes for the simulation and Lagrangian fluid interaction. The table 4.2 shows some of the main attributes contained in the FluidParticle class. Most of the attributes, such as the surface tension and the interface tension, will be explained in more details in the implementation chapter and their usage will become clearer.

  While a particle object is mainly used as part of the fluid simulation, it is also used as a rigid body sphere that interacts with the fluid and the environment (boundary and other rigid bodies).

- Environment
  This class manages the environment interaction with the fluid. It creates all the rigid bodies, manages their movements and handles all collision detection and resolution between the boundary, rigid bodies and the fluid particles.

Attribute	**Type**	**Use**
Position	Vector	Cartesian coordinates
Velocity	Vector	For particle movement
Mass	Float	Fixed at creation time
Net Force	Vector	Accumulated force
Acceleration	Vector	From Newton's second Law
Rest Density	Float	Pressure force calculation
Density	Float	Density at current position
Pressure	Float	Pressure at current position
Gas Constant	Float	Used to calculate pressure
Pressure Force	Vector	Navier-Stokes pressure force
Viscosity Force	Vector	Navier-Stokes viscosity force
Surface Tension Force	Vector	Smooths surface
Interface Tension Force	Vector	Multiple fluids interaction
Gravity Force	Vector	Gravitational free fall
Viscosity constant	Float	Calculation of viscosity force
Surface tension coefficient	Float	Calculation of surface tension
Surface tension threshold	Float	Calculation of surface tension
Interface tension coefficient	Float	Calculation of interface tension
Interface tension threshold	Float	Calculation of interface tension
Surface colour coefficient	Float	Calculation of surface tension
Interface colour coefficient	Float	Calculation of interface tension
Radius	Float	Collision handling
Colour	Colour	Visualisation

Figure 4.2: Attributes of the FluidParticle, their data type and usage

- Capsule
  A capsule is one of the rigid bodies, having the shape of the cylinder. While spheres move around, a capsule rotate around a fixed pivot and serves to mix the fluids that interact with it. It is often referred to as the mixer in later chapters.

- Cache and CacheItem
  The Cache class is responsible for the export of the simulation data and state to external files that could be imported to an external package, such as Houdini, and rendered or manipulated. It saves cache items, which are basically the fluid and the environment, at a user-defined sampling interval. The implementation chapter discusses about it in more details.

The other classes perform miscellaneous functions around the above classes and their functionalities are obvious from their names.

4.3 Simulation Pipeline

A fluid contains thousands of particles and layering and calculating their initial positions manually for the simulation is too daunting and complicated a task. Steele et al. (2004) and Priscott (2010) discuss the use of external models and using their vertices as initial positions for the particles.
Using this method makes it very easy to create fluids of various shapes and resolutions, such as spheres and cubes, or even real-life models such as a house or a creature, in an external package such as Maya or Houdini. These can then be exported to a standard *obj* format and loaded in the simulation engine.
The simulation would primarily be visualised using particles, either OpenGL points or sphere primitives. While this serves as an interactive pre-visualisation, advanced rendering is more desired. Since this is not part of the scope of this project, functionality will be implemented to export the simulation data to an external format such as *obj*, which could then be easily rendered using an external application such as Maya or Houdini.

Chapter 5

Implementation

5.1 Simulation Algorithm

A basic simulation step typically calculates the density, pressure, then the various forces such as pressure, viscosity, surface tension, interface tension and gravity. The acceleration is then integrated to get the next velocity and position. This is finally followed by collision detection and resolution, before the particle is displaced.

Algorithm 1: Algorithm to perform simulation of fluid

```
Load and create fluid particles
Initialise neighbour search structure
while timer ticks do
    Refresh neighbour search structure
    foreach particle $P$ in particle list do
        $L \leftarrow$ Get neighbours of $P$
        $\rho \leftarrow$ Calculate density of $P$ by iterating through $L$
        $p \leftarrow$ Calculate pressure of $P$
    end
    foreach particle $P$ in particle list do
        $L \leftarrow$ Get neighbours of $P$
        $F^{pressure} \leftarrow$ Calculate pressure force for $P$ by iterating through $L$
        $F^{viscosity} \leftarrow$ Calculate viscosity force for $P$ by iterating through $L$
        $F^{surface} \leftarrow$ Calculate surface tension force for $P$ by iterating through $L$
        $F^{interface} \leftarrow$ Calculate interface tension force for $P$ by iterating
        through $L$
        $F^{gravity} \leftarrow$ Calculate gravitational force on $P$
        $F^{net} = F^{pressure} + F^{viscosity} + F^{surface} + F^{interface} + F^{gravity}$
        $a = F/\rho$
        Integrate $a$ to get velocity $v$ and position $x$
        Apply collision detection and resolution on $P$ against environment
        Move $P$ and render it
    end
end
```

5.2 Fluid Creation

As mentioned in the design chapter, the fluid is created from an external model that imported from a file. Model creation has been done in Houdini using primitives such as poly sphere or poly box. Since this is not enough to provide the required number of vertices for a proper simulation, the *pointsfromvolume* sop has been used to fill the primitives with points/vertices. The parameter tab even allow specifying the number of points, thus the number of particles that form the simulation eventually can be easily controlled, and thus its resolution.

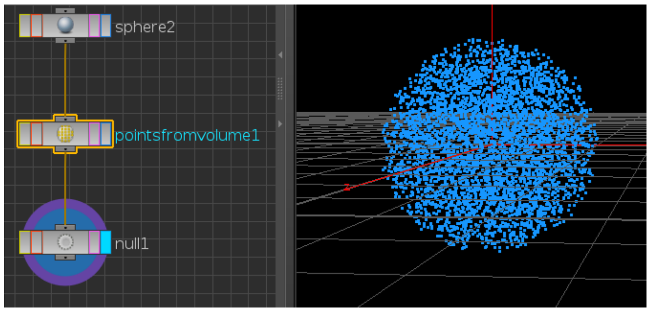


Figure 5.1: Fluid model creation in Houdini

The final model is exported to an *obj* file and loaded using the NGL graphics library (Macey, 2010). The mass of the particles is fixed at creation time and calculated as follows (Kelager, 2006):

$$mass = density * \frac{volume}{particlecount} \tag{5.1}$$

Algorithm 2: Algorithm for creation of the fluid particles

```
Load model
Read rest density $\rho_0$ and volume $V$ from configuration file
Get vertex count $n$ from model
Calculate mass $m = \rho_0 * V/n$
foreach vertex $v$ do
    Create particle, with $mass = m$, $position = v$
    Add particle to solver particle list
end
```

5.3 Force Calculation

This section describes in details the implementation of the various forces, which is the essential part of the fluid solver and that uses SPH as its base. The pressure and viscosity forces, which are directly obtained from the Navier-Stokes equations, have been discussed briefly in the Technical Background chapter. Apart from these two basic forces, two additional forces, surface tension and interface tension, are also discussed. This section also makes constant reference to the three smoothing kernels that were discussed in the Technical Background chapter.

5.3.1 Pressure and Density

The density, and consequently, the pressure must be evaluated at every particle's position as it evolves throughout the simulation. The pressure is computed using a modified version of the ideal gas state equation that was proposed by Desbrun and Cani (1996) and used by Müller et al. (2003), as follows:

$$p = k(\rho - \rho_0) \tag{5.2}$$

where ρ_0 is the rest density of the fluid and k is the gas constant.

This equation introduces a spring-like behaviour to the pressure calculation; while a density higher than the rest value produces a positive pressure that pushes the particles away, a lower density will result in a negative pressure and brings the particles together. The gas constant k depends on the temperature of the fluid and determines how strongly the pressure difference in the fluid is equalised.

The density at position $\mathbf{r}$ is calculated using SPH, as follows (Kelager, 2006):

$$\begin{aligned} \rho_i &= \sum_j \rho_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \\ &= \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \end{aligned} \tag{5.3}$$

Once, the pressure and density is computed, the pressure force is calculated by using SPH and the Poly6 smoothing kernel (3.12) (3.13) (3.14). The following equation resolves the non-symmetrical force issue (Kelager, 2006) and is used in this project:

$$\mathbf{f}_i^{pressure} = -\rho_i \sum_{j \neq i} \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) m_j \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{5.4}$$

Since the pressure and density must be found for all particles prior to the calculations of the forces, two loops must be used as shown in the simulation algorithm in the previous section; one to calculate the pressures and densities, then the other to calculate the actual forces.

5.3.2 Viscosity

The viscosity force acts not only as a damping force to particles, but also provides stability to the simulation. As mentioned earlier, only using SPH gives rise to a non-symmetric force; Müller et al. (2003) proposes the use of velocity difference instead of absolute values to solve this issue as follows:

$$\mathbf{f}_i^{viscosity} = \mu \sum_{j \neq i} (\mathbf{v}_j - \mathbf{v}_i) \frac{mj}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{5.5}$$

where v is the velocity of the particles.

5.3.3 Surface Tension

The surface tension force is an external force that is not from the Navier-Stokes equation and acts only on particles at the surface of the fluid. Its implementation is discussed in more details in this section.

Molecules inside a fluid are held in perfect balance through intermolecular attraction that is equal in all directions. However, this is not the case for particles at the surface of the fluid and this imbalance gives rise to the surface tension force. The latter acts along the surface normal in an inward direction towards the fluid and aims to reduce the curvature of the surface and thus, smooths the fluid surface (Müller et al., 2003).

The following formulas have been adapted from the paper of Kelager (2006).A colour field c is used to identify the surface of the fluid. Each particle is given a colour quantity of 1, while air is given a value of 0. It is calculated using SPH as follows:

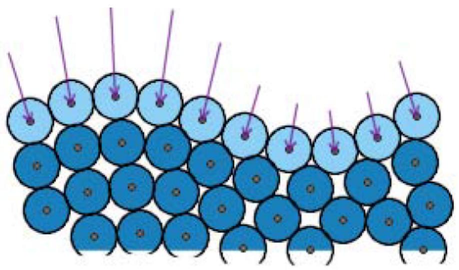


Figure 5.2: Behaviour of the surface tension force acting along the inward surface normals, in the direction towards the fluid (Kelager, 2006).

$$\begin{aligned} c_i &= c(\mathbf{r}_i) \\ &= \sum_j c_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \\ &= \sum_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \end{aligned} \tag{5.6}$$

Its gradient **n** gives the surface normal and the length of the gradient determines how close the particles are to the surface. It is calculated using SPH as follows:

$$\begin{aligned} \mathbf{n}_i &= \nabla c(\mathbf{r}_i) \\ &= \sum_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \end{aligned} \tag{5.7}$$

Its Laplacian calculates the surface curvature and is calculated as follows:

$$k = -\frac{\nabla \mathbf{n}}{||\mathbf{n}||} = -\frac{\nabla^2 c}{||\mathbf{n}||} \tag{5.8}$$

Finally, the surface tension force is calculated as follows:

$$f_i^{surface} = \sigma k_i \mathbf{n}_i = -\sigma \nabla^2 c_i \frac{\mathbf{n}_i}{||\mathbf{n}_i||} \tag{5.9}$$

The tension coefficient σ is different for the interaction of different fluids and it influences how strongly the surface holds itself. The Poly6 kernel (3.12) (3.13) (3.14) is used for the calculations. Moreover, the length of the gradient n would lead to instabilities in the force calculation as it nears zero; as such, the force is only calculated when the gradient length exceeds a threshold value β, else it is taken as zero (Kelager, 2006).

Algorithm 3: Algorithm for the calculation of the surface tension force

```
foreach $particle\ P\ in\ particle\ list$ do
    $gradient \leftarrow vector(0)$
    $laplacian \leftarrow 0$
    $L \leftarrow$ Get neighbours of $P$
    foreach $neighbour\ N\ in\ neighbour\ list\ L$ do
        $m \leftarrow$ mass of $N$
        $\rho \leftarrow$ density of $N$
        $gradient \leftarrow gradient + ((m/\rho) * W_{poly6gradient}$
        $laplacian \leftarrow laplacian + ((m/\rho) * W_{poly6laplacian}$
    end
    if $length(gradient) > \beta$ then
        $F^{surface} \leftarrow -\sigma * \text{normalise}(gradient) * laplacian$
    end
    else
        $F^{surface} \leftarrow 0$
    end
end
```

5.3.4 Interface Tension

The interface tension force is another additional external force that was first suggested by Müller et al. (2005) for the interaction of multiple fluids. The interface force acts perpendicular to the interface between the two fluids to minimise its curvature.
It is an adaptation of the surface tension force, discussed in the previous section, with the only difference being the colour field. Here, the interaction of polar and non-polar fluids is considered. Polar fluids tend to mix with polar fluids and are given a colour value -0.5; on the other hand, a polar fluid doesn't mix with a non-polar one and the latter bear a colour value of $+0.5$ (Müller et al., 2005).
The calculation of the gradient and Laplacian is similar to that for the surface tension, except that it uses the new colour values 0.5 and -0.5.

5.3.5 Gravity

The gravitational force is yet another external force that is caused by the acceleration due to gravity, called free fall, and it acts in the negative y direction. It has a value of $(0, -9.8, 0)$. Other external forces follow suit and can be implemented by a simple vector. For example, the addition of a vector $(5, 0, 0)$ to the accumulation of forces could be interpreted as a wind blowing in the positive x direction.

5.4 Integration

The acceleration of the particles is computed by dividing the net force, discussed in the previous section, by their density. The next step is to integrate the acceleration to find the new velocity and position of the particle. Two integration methods have been implemented in this project, namely, semi-implicit Euler and leap-frog.

5.4.1 Semi-Implicit Euler

While the explicit Euler method is the simplest integration method, it suffers from serious instabilities if the integration time-step is kept large. The semi-implicit Euler method, which is a slight derivative of the explicit Euler, is implemented in this project. The latter also suffers from numerical instabilities with big time-steps; however, it is known to conserve energy and thus is slightly more accurate. It is calculated as follows:

$$v_{i+1} = v_i + a \cdot dt \tag{5.10}$$

$$x_{i+1} = x_i + v_{i+1} \cdot dt \tag{5.11}$$

5.4.2 Leapfrog

The Leapfrog integration is a second order method that is more accurate than the first-order Euler method. It calculates velocities and positions at interleaved times, with the velocities calculated at half times and the positions calculated at full integer times. The velocity and position are calculated as follows (Priscott, 2010):

$$v_{i+1/2} = v_{i-1/2} + a \cdot dt \tag{5.12}$$

$$x_{i+1} = x_i + v_{i+1/t} \cdot dt \tag{5.13}$$

The initial velocity offset $v_{-1/2}$ is calculated using the Euler method as follows:

$$v_{-1/2} = v_0 - \frac{1}{2} \cdot dt \cdot a_0 \tag{5.14}$$

In this project, the leap-frog method has been implemented using a different set of formulas (Hut and Makino, 2004), which are similar to the previous formulas but have been rearranged and rewritten to calculate the velocity and position at full time steps. The derived formulas are as follows:

$$v_{i+1} = v_i + \frac{(a_i + a_{i+1}) \cdot dt}{2} \tag{5.15}$$

$$x_{i+1} = x_i + v_i \cdot dt + \frac{a_i \cdot dt^2}{2} \tag{5.16}$$

The velocity is calculated from the average of the current acceleration and next acceleration, therefore providing more stability to the next displacement of the particles.

5.5 Neighbour Search and Spatial Hashing

The search of neighbours and their contributions is a vital part of SPH; it is performed at every time-step of the simulation and directly impacts the accuracy of the SPH calculations as well as the speed of the simulation. A naive approach of checking every particle against all other particles results in a complexity of $O(n^2)$. This is most undesirable since it limits the number of particles that can be simulated. Also, the smoothing kernels of SPH use a core radius h beyond which, all other particles have no contribution to the current particle and searching them results in a waste of CPU resource. Several algorithms exist to find neighbours efficiently and fall under the *Nearest Neighbour Search* algorithms. One such very efficient algorithm which is suited to SPH fluid simulation is spatial hashing, developed by Teschner et al. (2003); this algorithm decreases the complexity from $O(n^2)$ to $O(nm)$, with m being the average number of numbers found and this can even be reduced with a more uniform distribution of particles (Kelager, 2006).

Spatial Hash Function
The spatial hashing method uses a hash function that converts a 3D position in space to a scalar hash key. This key is then used as an index to a hash-map to store particles in cells, as well as to retrieve neighbouring particles.

The hash function is the heart of this method, and it has been designed to prevent the particles that are significantly apart to be hashed to the same cell (Priscott, 2010). Kelager (2006) defines it as follows:

$$hash(\hat{\mathbf{r}}) = (\hat{\mathbf{r}}_x p_1 \text{ xor } \hat{\mathbf{r}}_y p_2 \text{ xor } \hat{\mathbf{r}}_y p_3) \mod n_H \tag{5.17}$$

A discretising function is used (Kelager, 2006) that divides the position by the cell size l, and returns the rounded integer values, as follows:

$$\mathbf{r}(\hat{\mathbf{r}}) = (\lfloor \mathbf{r}_x/l \rfloor, \lfloor \mathbf{r}_y/l \rfloor, \lfloor \mathbf{r}_z/l \rfloor)^T \tag{5.18}$$

The cell size l is the smoothing length used in our SPH kernel calculations.

n_H is the hash map size and is calculated as follows:

$$n_H = prime(2n) \tag{5.19}$$

Where n is the particle count and the function $prime(x)$ gives the next prime number $\geq x$.

The last missing piece to the hash function equation is the values of p_1, p_2 and p_3. These are large prime numbers and are suggested by Kelager (2006) as follows:

$$p_1 = 73856093, p_2 = 19349663, p_3 = 83492791 \tag{5.20}$$

Insertion in Hash Map
Now that the hash function is defined, the insertion of the particles and the filling in of the hash map is to be done. This is executed before the calculation of the forces.

Algorithm 4: Algorithm for the insertion of particles in the hash map

```
Clear hash map
foreach particle P in particle list do
    $x \leftarrow$ Get position $P$
    $\hat{x} \leftarrow$ discretise($x$)
    $hashKey \leftarrow$ hashFunction($\hat{x}$)
    Insert $< hashKey, P >$ in hash map
end
```

Searching of Neighbours
Once the hash map is refreshed, the neighbours of a particle can be searched by first generating the hash key from its position, then retrieving all particles from the map with the generated hash key as an index. This, however, is not enough since not all neighbouring particles are hashed to the same cell. Kelager (2006) also suggests the use of a bounding box, so that the search area is extended

around the particle. The bounding box limits for a particle at position r are defined as follows:

$$BB_{min} = \hat{\mathbf{r}}(\mathbf{r} - (h,h,h)^T)\ ,\ BB_{max} = \hat{\mathbf{r}}(\mathbf{r} + (h,h,h)^T) \tag{5.21}$$

A bounding box is used since it is very easy to calculate its limits and iterate through it; however, only particles within the distance of the smoothing length from the searched position is required, as follows:

$$||\mathbf{r} - \mathbf{r}_j|| \leq h \tag{5.22}$$

Algorithm 5: Algorithm used to find the neighbours for a particle (adapted from Priscott (2010))

```
Input: A particle $P$ with position $X$
1. get particles in the same cell as particle P
Generate hash key $key$ using position $X$
Get all particles in the map with the generated hash key $key$
Add to neighbour list
2. define bounding box and search particles within that
Define bounding box limits $BBMin$ and $BBMax$
Discretise $BBMin$ and $BBMax$
for $BBMin$ to $BBMax$ do
    Create a Cartesian position $c$ within the bounding box
    Generate hash key $key2$ for the created position $c$
    Search for particles from the map having the generated hash key $key2$
    if they are not duplicates then
        if they are within smoothing distance $h$ then
            then add them to neighbour list
        end
    end
end
```

5.6 Environment Interaction

The environment consists of the boundary and rigid bodies. These interact with the fluid particles and modify their behaviour. Kelager (2006) suggested the use of implicit functions to model rigid bodies such as spheres and capsules and this has been implemented in this project. While an implicit function doesn't give simply control over the shapes of the rigid bodies, it helps keep the computations less complex because implicit surfaces can be defined very easily with few single mathematical functions. The collision handling mechanism, implemented in this project, assumes that the environment is fixed such that the fluid particles do not influence them; while this is not true in real-life and violates Newton's third law of motion of action-reaction, it keeps the collision handling easier and less-computationally intensive.

5.6.1 Collision Detection and Handling

Collision detection is performed at every time-step for all the particles. It simply involves solving the implicit function at the current position of the particles; a negative result from the function, $F(x) < 0$ implies that collision has occurred.

Once collision is detected, the following needs to be determined for the resolution, as follows:

- the contact point **cp** where the particle penetrate the obstacle's surface,
- the penetration depth d that is given by the value of the solved implicit function, and
- the surface normal **n** of the collided surface.

The calculation of the contact point **cp** is not so trivial since it is calculated from the centre of the rigid body along the normal. Divergence occurs between the true contact point and the calculated point, as shown in the figure 5.3, however this is minimised when the time-step is small.

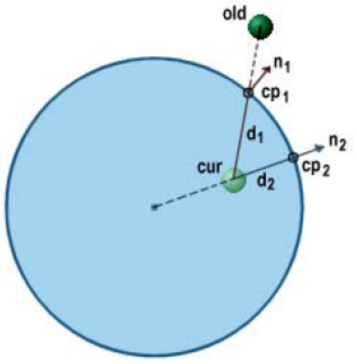


Figure 5.3: Two possible collision determinations from the same particle position update with $\mathbf{cp}_1$ being the correct point and $\mathbf{cp}_2$ the calculated one. (Kelager, 2006)

Collision resolution is a very active research topic in Computer Graphics and several methods and algorithms have been developed and proposed by many papers. Monaghan (1994) developed a repulsive force with the LennardJones potential, called boundary force. Bell et al. (2005) introduces penalty force-based to resolve collision. Impulse-based collision response is a very popular technique that modifies the velocity of the particle at the time of collision. In this project, the hybrid impulse-projection method proposed by Kelager (2006) is used, which simply projects the collided particle back to the surface along the surface normal.

This new position $\mathbf{r}_i$, in the case of our implicit surfaces, is simply the contact point:

$$\mathbf{r}_i = \mathbf{cp} \tag{5.23}$$

The velocity is reflected back along the surface normal. A restitution coefficient, c_R such that $0 \leq c_R \leq 1$, is used to determine the amount of the velocity reflected back, thus mimicking the loss of energy during a collision, with a value of 0 giving rise to an inelastic collision and a value of 1 simulating a perfectly elastic situation (Kelager, 2006). The formula to calculate the velocity is as follows:

$$\mathbf{v}_i = \mathbf{v}_i - (1 + c_R)(\mathbf{v}_i \cdot \mathbf{n})\mathbf{n} \tag{5.24}$$

5.6.2 Environment Objects

5.6.2.1 Boundary

The boundary is implemented using a primitive bounding box. This is the simplest primitive and allows for really cheap calculations to detect and resolve collision.

The box is defined as a collection of 6 faces, each representing a limit *XMin*, *XMax*, *YMin*, *YMax*, *ZMin* and *ZMax.* The x, y and z components of the particle's position is simply checked against these limits. In case of collision, they are set to these limit values.

The new velocity follows the same line as the formula discussed in the previous section. However, since the latter involves dot product and vector multiplication, it is a bit costly. In the case of a bounding box, the surface normals are the unit normals. Thus, the velocity is simply reflected as follows:

$$v_i = v_i * -c_R \tag{5.25}$$

5.6.2.2 Periodic Wall Boundary

A periodic wall is also implemented in this project. This is an extension of the boundary box, discussed in the previous section and as such, has the same collision handling mechanism.

A periodic *sine* function is used to move the right wall, found at the maximum x direction, up to a maximum amplitude of a to and from its initial position p_0.

$$a \sin(b\theta) \tag{5.26}$$

Increasing the value of a will make the wall have larger displacements around its rest position. The angle θ of the *sine* function can also be multiplied by a scale factor b to increase or decrease the period of the function and thus, influence the speed of the wall movement.

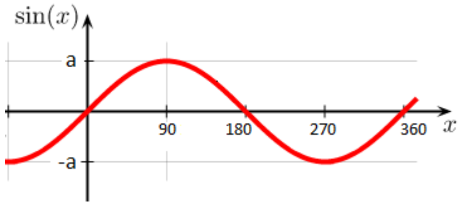


Figure 5.4: sine wave with amplitude a and $b = 1$. Adapted from `http://commons.wikimedia.org/wiki/Image:Sine.svg`

5.6.2.3 Rigid Bodies

As mentioned initially, the rigid bodies are modelled with implicit functions, which are easy to define and allow simple collision detection mechanism. Two rigid bodies have been implemented in this project, a sphere and a capsule.

Sphere
A sphere is the simplest implicit surface to define, using the following formula (Kelager, 2006):

$$F_{sphere}(x) = ||x - \mathbf{c}||^2 - r^2 \tag{5.27}$$

where $\mathbf{c}$ is the centre of the sphere and r its radius.

The contact point $\mathbf{cp}$ is calculated as follows:

$$\mathbf{cp}_{sphere} = \mathbf{c} + r\frac{x - \mathbf{c}}{||x - \mathbf{c}||} \tag{5.28}$$

The penetration depth d is given by:

$$d_{sphere} = |||\mathbf{c} - x|| - r| \tag{5.29}$$

where $|a|$ gives the absolute value of a.

The unit surface normal $\mathbf{n}$ is calculated using the following formula:

$$\mathbf{n}_{sphere} = \text{sgn}(F_{sphere}(x))\frac{\mathbf{c} - x}{||\mathbf{c} - x||} \tag{5.30}$$

Spheres are free to move and interact with the fluids, boundary, capsules and other spheres. They are also subject to gravitational forces and act as a full rigid body within the container.

Capsule

A capsule is another implicit surface that is widely used in computer graphics. While easily defined as an implicit mathematical formula, it can mimic the usage of a cylinder or a box. In this project, it is fixed to a pivot position, with a z-orientation and it can rotate freely around the y-axis, thus simulating a mixer.

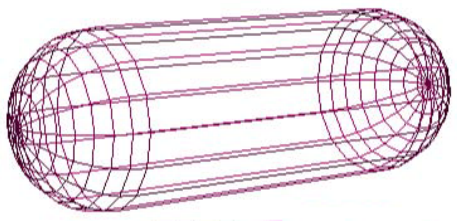


Figure 5.5: A capsule in wireframe made up of 2 hemispheres and a cylinder. (Kelager, 2006)

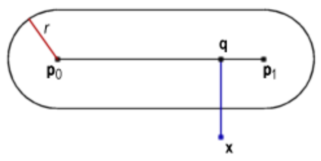


Figure 5.6: A capsule defined by the two end points $\mathbf{p}_0$ and $\mathbf{p}_1$, and a radius r. (Kelager, 2006)

A capsule is defined using the following formula:

$$F_{capsule}(x) = ||\mathbf{q} - x|| - r \tag{5.31}$$

where

$$\mathbf{q} = \mathbf{p}_0 + \left(\min\left(1, \max\left(0, -\frac{(\mathbf{p}_0 - x) \cdot (\mathbf{p}_1 - \mathbf{p}_0)}{||\mathbf{p}_1 - \mathbf{p}_0||^2}\right)\right)\right)(\mathbf{p}_1 - \mathbf{p}_0) \tag{5.32}$$

The contact point **cp** is calculated as follows:

$$\mathbf{cp}_{capsule} = \mathbf{q} + r\frac{x - \mathbf{q}}{||x - \mathbf{q}||} \tag{5.33}$$

The penetration depth d is given by:

$$d_{capsule} = |F_{capsule}(x)| \tag{5.34}$$

The unit surface normal **n** is calculated as follows:

$$\mathbf{n}_{capsule} = \mathrm{sgn}(F_{capsule}(x))\frac{\mathbf{q} - x}{||\mathbf{q} - x||} \tag{5.35}$$

A **sgn** function is used in the calculation of the surface normals; this simply gives the sign of the function, positive or negative.

5.7 Particle Injection

Functionality was implemented to allow the injection of particles into the fluid. As with the fluid model, the injected particles also are loaded from an external model and the latter typically has a reduced particle count. A list of particle prototypes is kept in a separate list; each one of those contains the basic properties for the fluid that it represents. Therefore, if there are 2 fluids loaded in the simulation, the list contains 2 prototypes.

Algorithm 6: Algorithm used to inject particles into the fluid simulation

```
Get index $i$ of current fluid for which particles are to be injected
Get particle prototype $p$ for the index $i$
foreach injected model vertex $v$ do
    Create a particle $Z$ with $position = v$ and $properties = p$
    Add $Z$ to list of particles forming the fluid
end
```

This approach allows the new particles to be added to the same list of particles that make up the simulation fluid. Therefore, once these particles are injected, they no longer differ from the fluid particles and get subjected to the same SPH forces as all others.

There is an option that is provided to specify the time at which these particles start behaving like fluid particles; this could be as soon as they are injected, or when they hit the boundary or some rigid body. Varying this gives some very interesting effects. Screenshots have been given in the *Simulation, Results and Evaluation* chapter

5.8 Visualisation and Data Exporting

5.8.1 OpenGL Rendering

As stated initially, specialised rendering techniques, such as raycasting, is not part of the scope of this project. The particles have been rendered as sphere primitives, using OpenGL. Each fluid is given a specific colour so that their interactive can be observed. The obvious disadvantage of this approach is that gaps appear in low-density regions. One approach, suggested by Kelager (2006) is to modify the radius of the sphere to increase or decrease according to the density and thus cover the gaps; this has not been implemented in this project as additional calculations are required and visualisation is not the primary focus of this project.

5.8.2 Exporting Simulation Data To Houdini

The ability to export simulation data and state is implemented in this project, so that better and enhancement rendering can be achieved. The export is specifically done to *geo* files, which is Houdinis native geometry ASCII format. This file format can be easily read and written to, thus the reason for choosing it for this project. Moreover, being native to Houdini, it is very easy to load in Houdini and per-frame animation is very fast.

While the contents of the *geo* file could contain all sort of combination between various attributes, primitives, groups and etc., a simple approach to export points has been adopted here. The latter is achieved by writing the set of points as homogeneous x, y, z and w coordinates, one per line. Once exported to a *geo* file, loading in Houdini is intuitive with the *file* sop. Point manipulation in Houdini is a very common task and lots of enhancements could be applied to particles to render them in multiple forms.
One very easy method is the use of the *copy* sop to copy spheres or other primitives onto the particles and thus have a simulation of spheres (Horvath and Illes, 2007). Houdini provides another very interesting sop called *particlefromfluid* which generates a surface from the simulation particles. Further enhancements can be done, such as the application of a procedural liquid shader. In brief, Houdini itself takes care of all the visualisation process; all it needs is the simulation data and this has been implemented in this project.

Apart from the fluid particles, the positions of the rigid bodies can also be exported. The rigid bodies could thus be reconstructed in Houdini with enhancements and behave according to the simulation data. The boundary box transformations are also exported, thus its periodic movements can be recreated in Houdini.

Chapter 6

Simulation, Results and Evaluation

This chapter focuses on the simulation, setup parameters and execution. Various scenarios are discussed and simulated to show the usability and flexibility of the simulation. The chapter ends on a discussion of some of the issues dealt with the simulation and efficiency considerations of the implemented solution.

6.1 Simulation Parameters

While it is desired to have the parameters of the simulation set to true real-life values, this is not always possible, either because the true values are too big for the simulation, such as the gas constant or they are too tiny such as in the case of the surface tension coefficient. Many of the parameters are left to the user for tweaking to achieve a stable simulation. Moreover, large viscosity and damping is required to take care of the various pressure shocks that would otherwise explode the simulation. Due to this, the range of viscosity values that could be simulated is really narrow. The mass of the particles has also to be set high so that the heaviness stabilise the simulation.

6.2 Scenarios and Results

Various scenarios have been setup and simulated, that includes the interaction of multiple fluids with the different rigid bodies, spheres and capsules, arranged in diverse ways. Some very interesting scene setups have been made and the results obtained are demonstrated in the following sections.

6.2.1 Single Fluid

Several scenarios were simulated, figure 6.1 and figure 6.2 and in all of the cases, 4 phases in the flow of the liquid flow were observed:

1. particles falling down and splashing on the floor,

2. particles moving away from the centre of the initial splash spot towards the sides of the boundary, due to the high density and induced pressure generated there,
3. particles collide against the boundary and slowly make their way towards the initial splash position, as they move from the boundary (high density) towards the centre of the initial splash (very low density), reflecting the workings of the pressure term in the Navier-Stokes formula as discussed in section 3.2,
4. particles come together at the initial splash position and again start moving away due to the high pressure created at that position.

6.2.2 Multiple Fluids

Particle-based simulations simplify the interaction of multiple fluids. Müller et al. (2005) demonstrates multiple fluid interaction using interface forces. Several tests were done to simulate the interaction of multiple fluids and some screen-shots are included in this section in figures 6.3 and 6.4.

6.2.3 Rigid Bodies Interaction

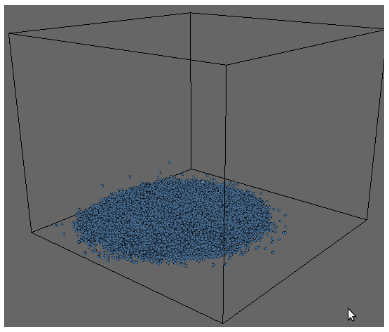
Several scenarios have been arranged with the implemented rigid bodies, sphere and capsule, and screenshots are displayed in figures 6.5, 6.6, 6.7, 6.8, 6.9, 6.10 and 6.11.

6.2.4 Channels

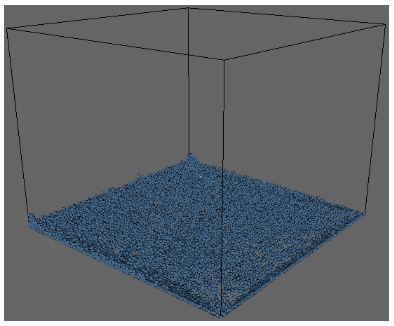
Although the rigid bodies, sphere and capsule, seem very basic, they can be grouped and arranged to simulate some very interesting situations; in the following tests, capsules have been used to simulate slanted surfaces and a very interesting funnel.

6.2.5 Periodic Wall

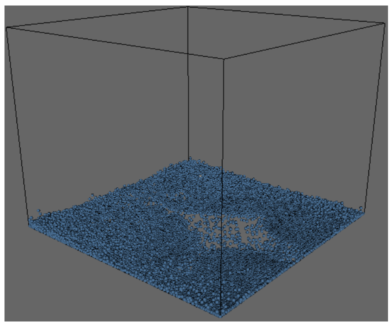
Another simulation has been done including the mixing of 2 fluids, but with the periodic wall enabled. The simulation results, figure 6.12, were similar to those obtained in the classic dam breaking problem, where the slowly moving wall ease the pressure contained inside the boundary box to give rise to some very interesting waves that goes high up in the air, then break down rapidly.



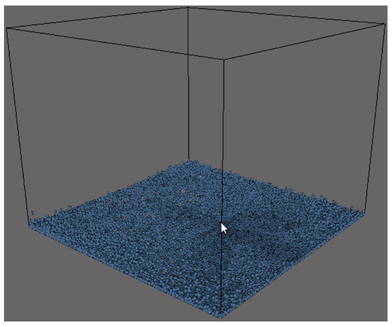
(a) Initial splash on the floor



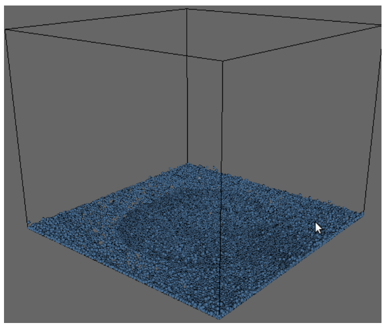
(b) Moving away towards boundary due to high pressure at centre



(c) Coming back due low pressure at centre

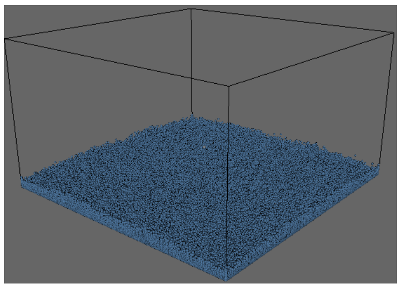


(d) High pressure induced at centre due to collision



(e) Pressure creates outward wave

Figure 6.1: Simulation of a single sphere fluid of 13566 particles


(a)

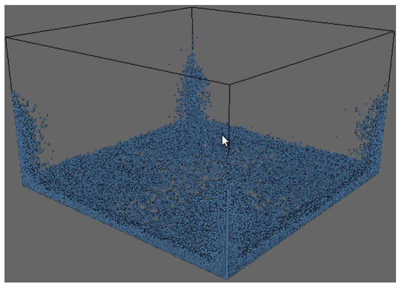
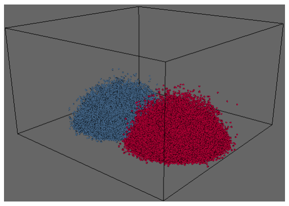
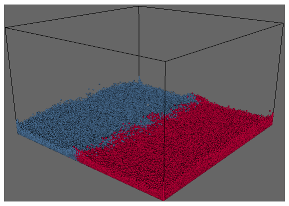
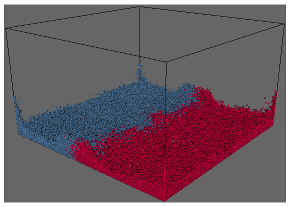
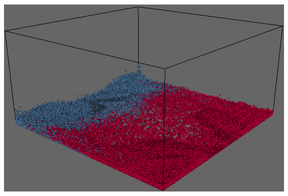

(b)

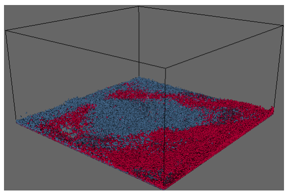
Figure 6.2: Simulation of a single fluid of 65536 particles


(a)


(b)


(c)


(d)


(e)

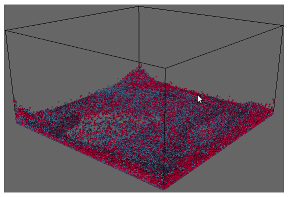

(f)

Figure 6.3: Simulation of 2 mixing fluids, with a total particle count of 103135

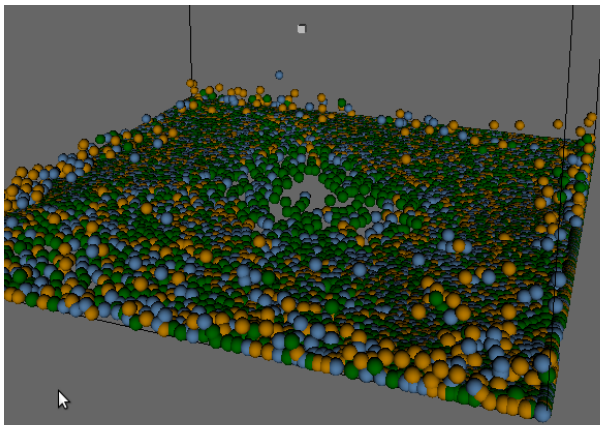


Figure 6.4: The mixing of 3 fluids

6.2.6 Particle Injection

Several simulations were done that incorporated the use of particle injection. One very interesting feature of the implementation of the particle injection is giving the ability for the particles to behave as particles only until they hit either the boundary or some rigid body, after which they start behaving as the other fluid particles. This made it possible to simulate hoses and water jets as shown in the figures 6.13 and 6.14.

6.3 Issues and Considerations

6.3.1 Time-Step

The integration time-step must be chosen very carefully because it directly impacts the stability of the simulation due the integration. While it is desired to keep it as big as possible, this would have repercussions on many other aspects of the simulation. Keeping everything stable, only a time-step of the order of 0.01 s has been achieved.

6.3.2 Compressibility

The gas constant, used in the calculation of the pressure (5.2) from the density, determines how strongly the pressure difference in the fluid is equalised. The aim is to make it as big as possible so that the liquid comes to rest fast after a pressure change and thus follow a near-incompressible behaviour that is expected of a liquid. However, this is not easy to achieve as it models a mass-spring behaviour

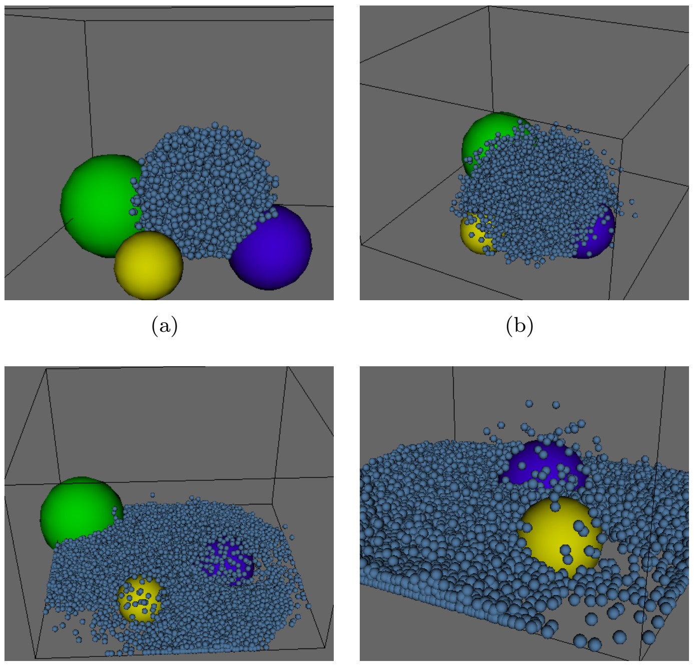
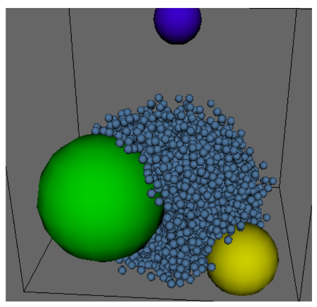
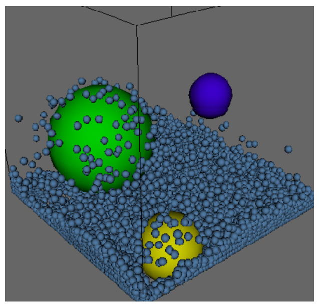
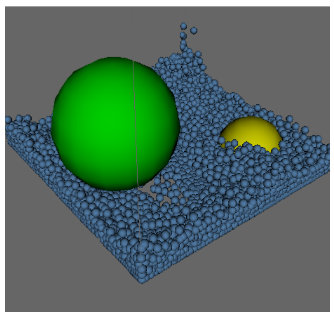

(a) (b) (c) (d)

Figure 6.5: Fluid interaction with sphere RBD


(a)


(b)


(c)

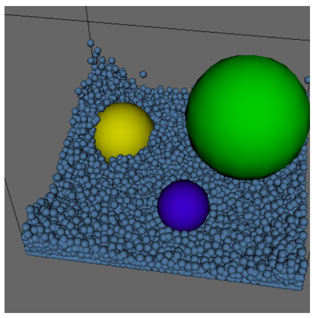
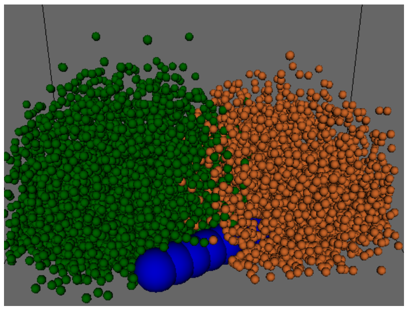
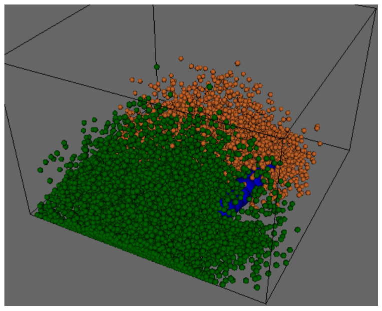

(d)

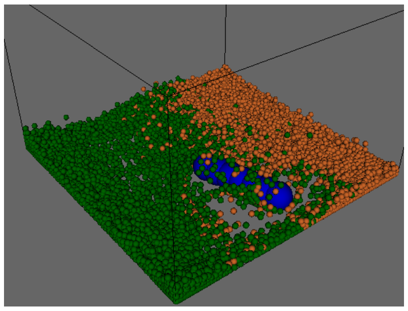
Figure 6.6: More fluid interaction with dynamic sphere RBD



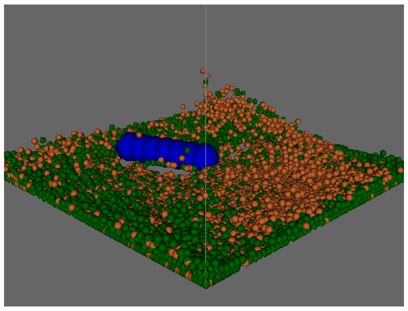
(a)



(b)

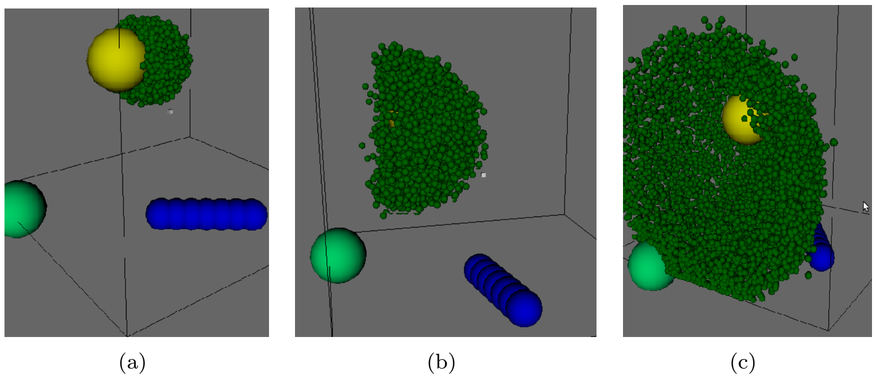


(c)



(d)

Figure 6.7: Fluid mixing with capsule RBD



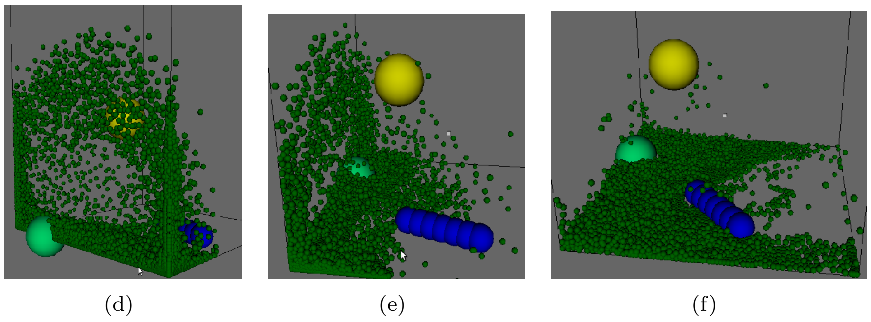


Figure 6.8: Fluid given an initial velocity and thrown onto a static sphere, before splashing on the wall

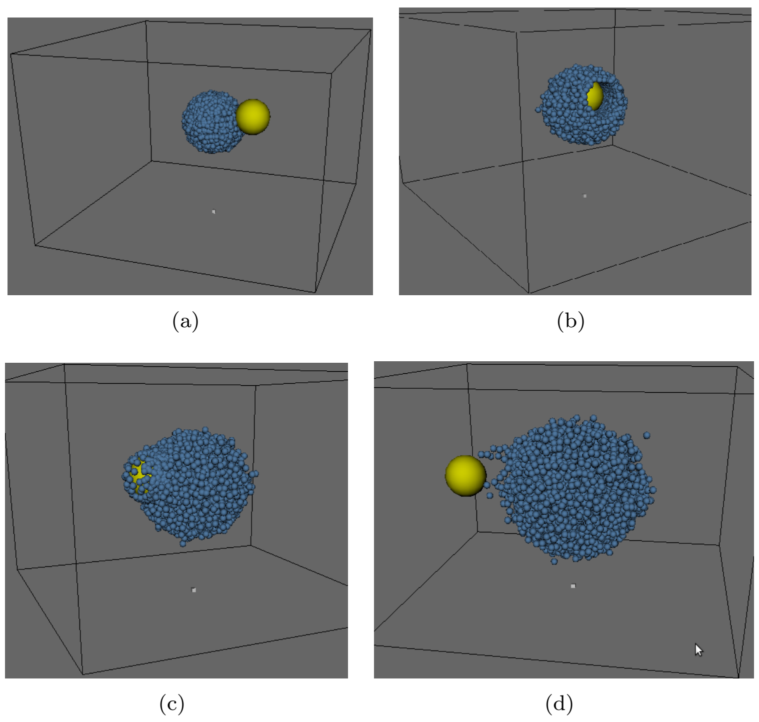


Figure 6.9: Dynamic Sphere simulating a bullet through the fluid

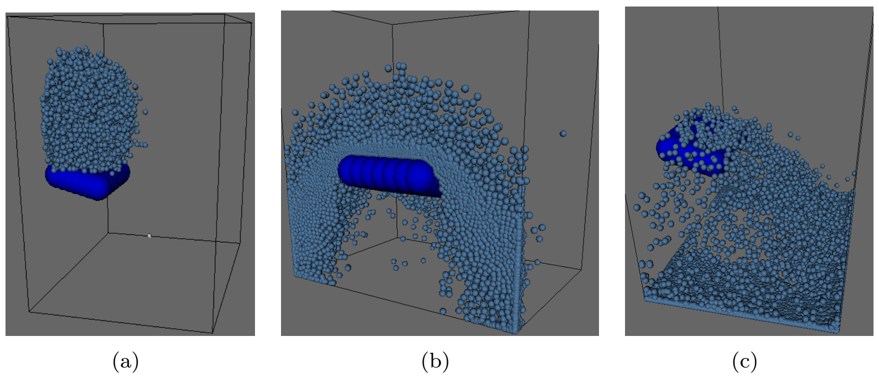


Figure 6.10: Static capsules arranged to simulate a slanted hard surface

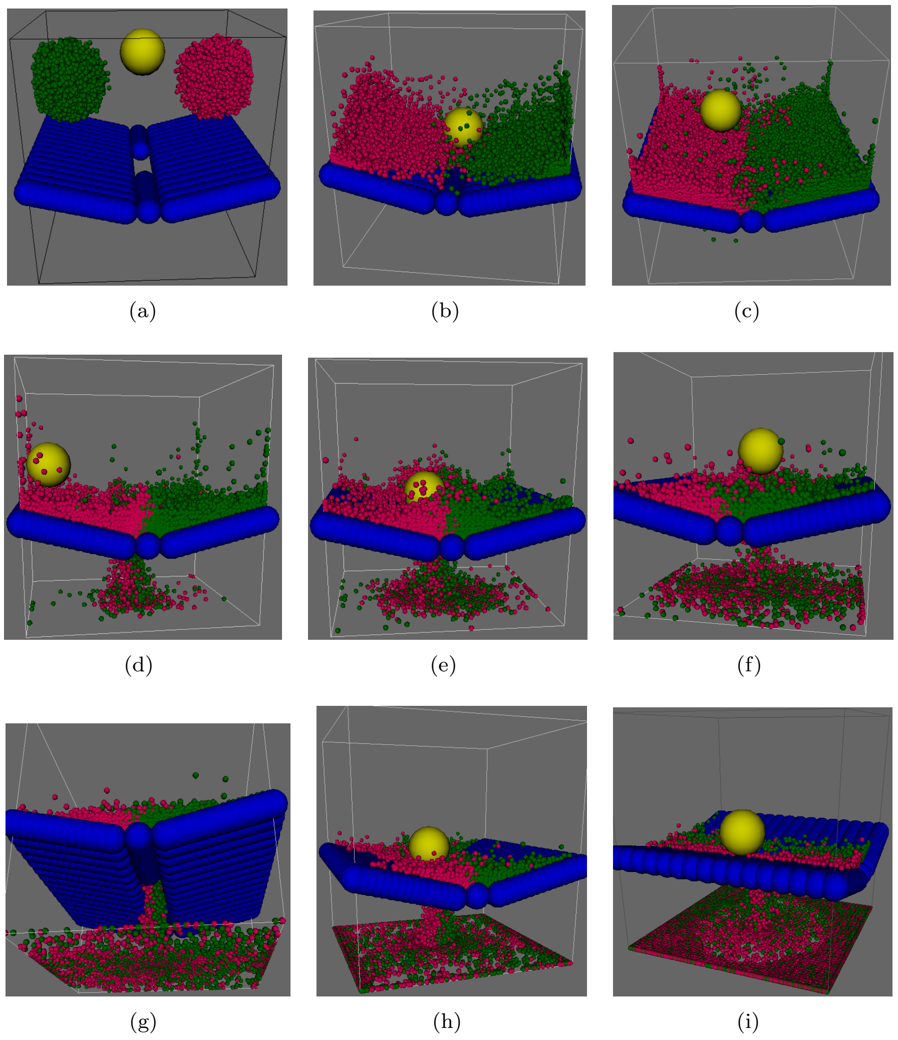
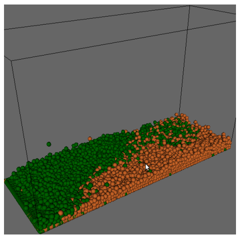
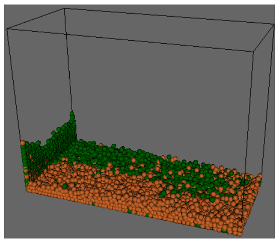


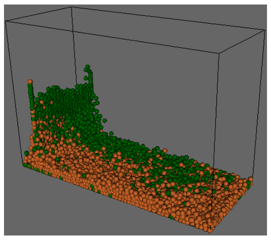
Figure 6.11: Use of static capsules arranged to simulate a funnel with a dynamic sphere blocking the funnel occasionally as it moves around



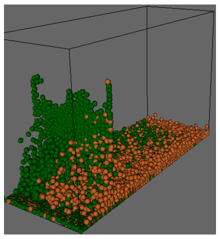
(a)



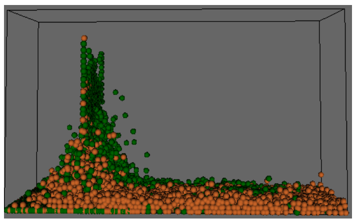
(b)



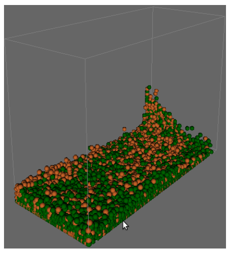
(c)



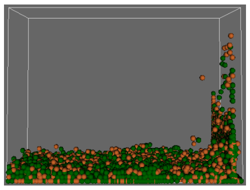
(d)



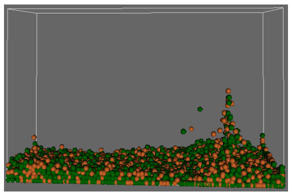
(e)



(f)



(g)



(h)

Figure 6.12: Periodic wall to simulate the classic dam breaking problem and mixing of 2 fluids

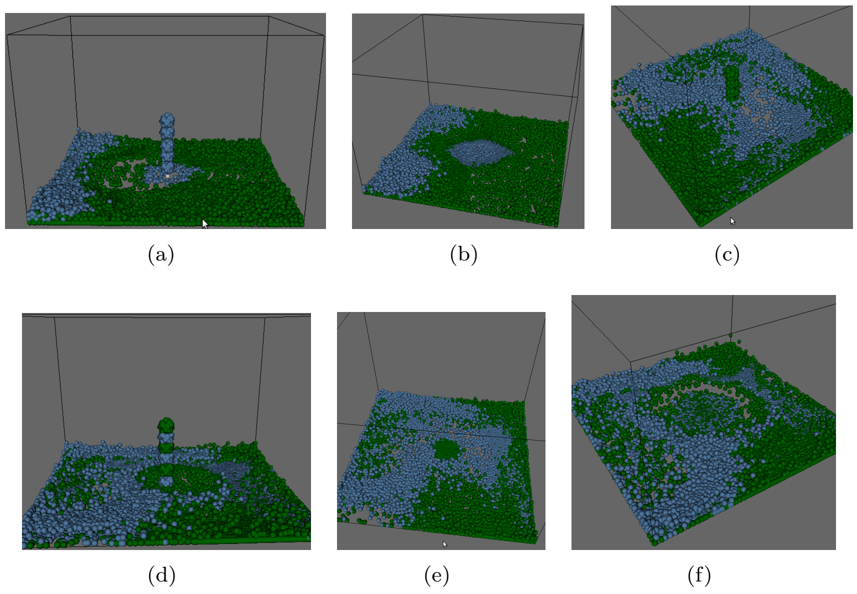


Figure 6.13: Injection of particles into simulation. figure (d) shows the ease with which the hose fluid can be changed to inject in particles of both of the fluids.

and a huge value will make it act like a really hard spring that will undergo huge numerical instabilities when a large change in pressure occurs (Kelager, 2006); in this case, the integration time-step must be decreased to a really small value to stabilise the system and this is not desirable as it reduces the interactivity of the simulation. A compromise has to be made between the gas constant and the time-step, such that both are kept as high as possible, while having a stable simulation. In this project, a gas constant of 10 with a time-step of 0.01 is used to achieve stable simulations.

6.3.3 Viscosity and Damping

The simulation contrasts with real-world liquids in that it appears to be over-damped. While this can be resolved by decreased the viscosity constant, it also implies making the integration time-step really tiny to prevent explosion of the simulation and this impacts the interactivity of the simulation, which is not desired. Also, mathematical damping introduced through viscosity keeps the simulation stable by preventing it from exploding whenever there is a huge pressure shock, thus decreasing the viscosity in this case in not a good alternative. To be able to keep the time-step as big as possible gives a really narrow margin of viscosity values in this project; the only other way to solve this prob-

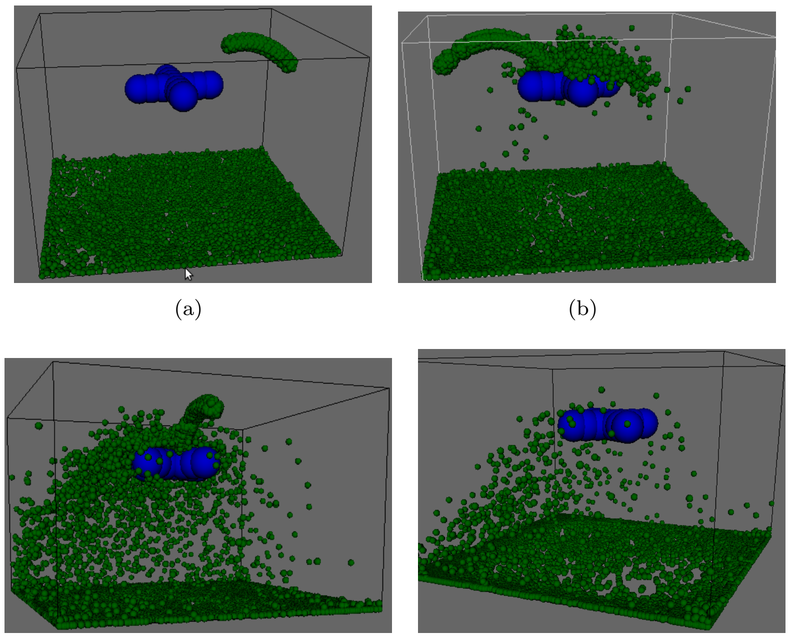

(a) (b) (c) (d)

Figure 6.14: Directional injection of particles into simulation with interaction with capsule mixer to simulate water jet and rain. The hose particle initially behave as particles only, until they first hit a rigid body, after which they start behaving as fluid particles.

lem is using implicit integration, but this process is very computation-intensive.

6.3.4 Smoothing Length

The smoothing length of the smoothing kernels determines the speed and accuracy of the SPH calculations and the overall stability of the simulation. The figure 6.15 shows the impact of various sizes of the smoothing length.

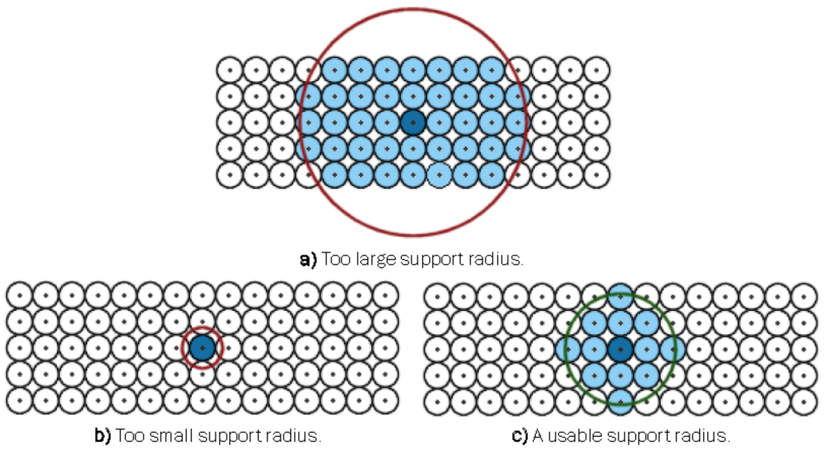

a) Too large support radius.
b) Too small support radius.
c) A usable support radius.

Figure 6.15: A 2D illustration of the problem of using either a) a too large support radius, or b) a too small support radius. The dark sphere in the centre is the particle in question. The support radius is illustrated as a circle. In this example the support radius in c) will be a good choice. (Kelager, 2006)

A huge smoothing length results in too many neighbours; this increases the complexity of the force calculations which is time-consuming, but most importantly, reduces the weights of the contributions of the neighbours to spread them over a bigger distance, such affecting the stability of the simulation (Kelager, 2006). On the other hand, a tiny smoothing length will results in very little number of neighbours and thus not enough contributions to compute the forces; the simulation will appear erratic and unstable.

Kelager (2006) suggests the following equation that could be used to determine the appropriate smoothing length:

$$h = \sqrt[3]{\frac{3\mathbf{V}x}{4\pi n}} \tag{6.1}$$

Where $\mathbf{V}$ is the volume and n, the particle count. This still leaves an unknown

x which is the number of neighbours desired. The aim would be to keep x small to reduce the calculation complexity, while still getting a stable simulation.

For the simulations done in this project, the smoothing length has been set as 0.3, which is roughly near to the result of the formula, described previously, when $x = 10$.

6.3.5 Performance And Efficiency

In this section, the performance of the solver was observed over various amounts of particles, in terms of memory and CPU usage. The aim is to get a linear relationship between these metrics and the particle count, so as to get as close as possible to a complexity of O(n).

6.3.5.1 First Approach to Neighbour Search optimisation

Among all the tasks that are executed in the simulation loop, the searching of neighbours for a particle is most complex and resource-intensive. This is due to the numerous mathematical operations, such as *mod* and *xor*, that need to be performed to hash the positions, iterate over a bounding box to get neighbours and check for duplicates. This routine need to be executed for every single particle in the simulation and at every iteration. The situation is worsened by the fact that the neighbours need to be calculated twice, as demonstrated in section 5.1, one for the calculation of the density and the other for the actual force calculation. Therefore, the optimisation of the neighbour search would result in a drastic performance gain.

Two methods of querying the neighbours are considered. The first one is to query the latter **on the fly**, i.e., get the neighbours at the time they are needed, thus inevitably calculating the neighbours twice. The second method is to get the neighbours of all the particles and save them to a **big neighbour list**, prior to all other tasks, then use the neighbours from this big list (which is a simple fetch from a list) for the density calculation and force calculation. This latter method, thus, eliminates the need for double calculations and results in a performance boost. Two simulations are run, one with a particle count of 13566 and the other with a particle count of 29667, over 3000 iterations using these two methods alternatively and observing their execution time (figure 6.16).

The graph 6.16 demonstrates the performance boost obtained by using the second method. While the execution time scales accordingly for the 13566 particle count over the 3000 iterations, it increases non-linearly as the number of particles is increased, i.e., for the 29667 particle count as from the 2000-mark. This

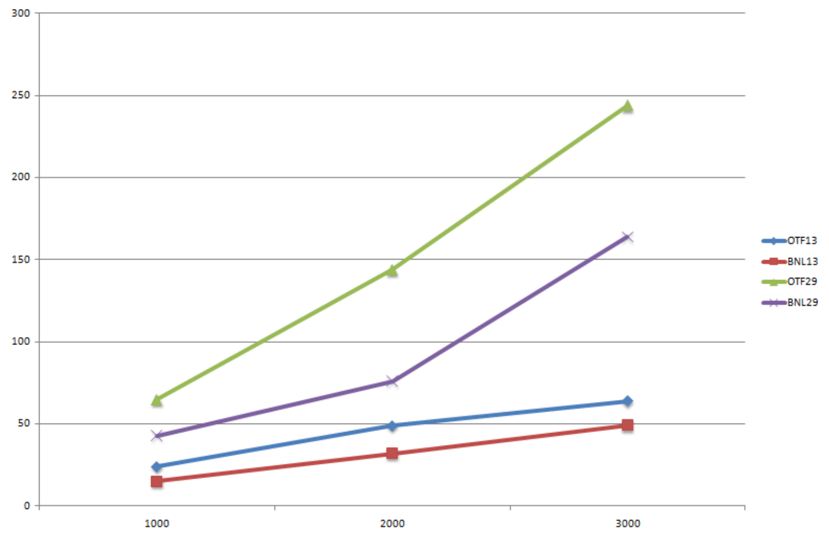


Figure 6.16: Execution time (in minutes) over iterations. OTF13 - on the fly with 13566 particles, OTF29 - on the fly with 29667 particles, BNL13 - big neighbour list with 13566 particles, and BNL29 - big neighbour list with 29667 particles.

is due to the fact that the increased particle count increases the number of neighbours with which each particle interacts as they come close to each other and this definitely increases the complexity of the force calculations; however, from gradient of the graph remains constant from the 2000-mark to 3000, thus proving a linear complexity.

On the other hand, the memory usage pattern is the total opposite; the *on the fly* method uses the least amount of memory as compared to the *big neighbour list* method, mostly due to the fact that the latter needs storage to keep the neighbours. Since, the neighbour list per particle is totally dynamic, space cannot be reserved in advance and thus the performance of the data structure used for the storage degrades with more iterations and particles. This is illustrated in the graph 6.17.

6.3.5.2 CPU usage optimisation using OpenMP

Although a big performance boost was obtained using the *big neighbour list* method, discussed in the previous chapter, the solver still is not fully optimised and as such cannot be used for big particle counts. The Lagrangian fluid simulation is basically a subset of particle systems, where the use of multiple cores is becoming very popular to reduce the execution time as the complexity increases.

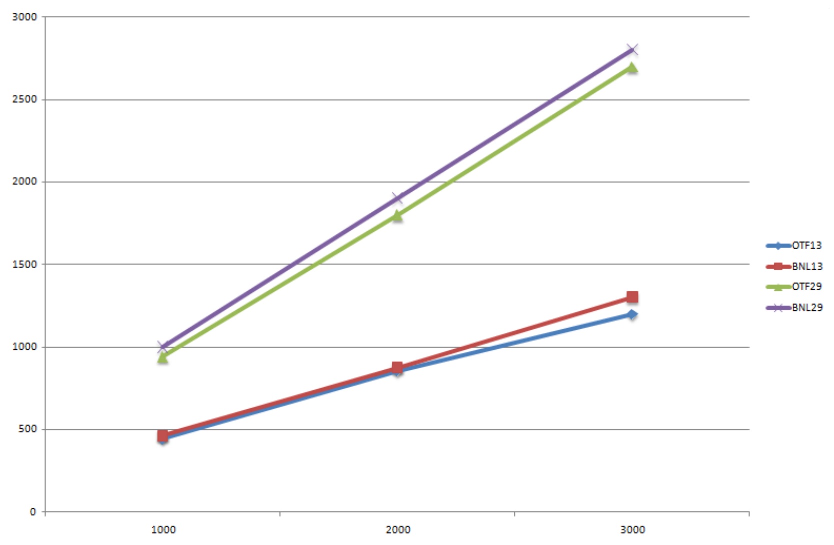


Figure 6.17: Memory usge (in Mb) over iterations. OTF13 - on the fly with 13566 particles, OTF29 - on the fly with 29667 particles, BNL13 - big neighbour list with 13566 particles, and BNL29 - big neighbour list with 29667 particles.

The use of OpenCL was initially investigated in this project, but failed to produce any performance gain. This could be explained by the fact that OpenCL works with buffers of bytes, whereas our implementation is completely object-oriented, thus requiring the need for the flattening of data to pass to the OpenCL kernels, then uncompressing them afterwards to update the objects. While a pure OpenCL implementation would definitely result in a highly optimised simulation, as demonstrated by numerous projects online, it takes away all the code-readability and maintainability provided through the use of classes and objects. Also, OpenCL kernels deal directly with memory and the optimisation process needs to consider issues such as memory bandwidth optimisation and minimising cache misses, thereby making the implementation hardware dependent.

On the other hand, multiprocessing using OpenMP has been implemented in this project and it provided a very significant performance boost. In the following sections, a brief overview of OpenMP is given, followed by a discussion of its implementation in this project.

6.3.5.2.1 Overview of OpenMP OpenMP is an API, defined by a group of computer and software vendors, which enables the development of parallel applications through the use of multi-threading and multi-processors. It supports C/C++ and Fortran across several platforms (Barney, 2011).

OpenMP is based on a shared-memory model (Adve and Gharachorloo, 1996), with multiple threads are created and allocated jobs and shared memory space is used to synchronise the result of their execution. As such, it uses the concept of *private* and *shared* memory variables and makes a clear distinction between these two. Care should be taken when using shared memory to prevent race conditions, with different threads overwriting the shared variable each in their turn and resulting in an undetermined state for the shared variable.

The major strength of OpenMP is that it uses preprocessor directives, in the form of *#pragma omp*, to mark out sections of code or procedures or tasks that should be executed in parallel. This makes it easy to integrate OpenMP at an advanced stage of a project, without disrupting the structure of the code and it also allows a mixture of sequential and parallel code, based on the fork-join model (Barney, 2011). It is very easy to disable the use of OpenMP and the code reverts back to its sequential form. Parallelism is provided at three different levels; *sections* that are small independent procedures, *loops* and higher-level *tasks*. In this project, *loop parallelism* has mostly been used, that offers parallelism in for-loops. It works by dividing the for-loop iterations among the threads and then, each thread executes its iterations independently of all others in parallel (figure 6.18).

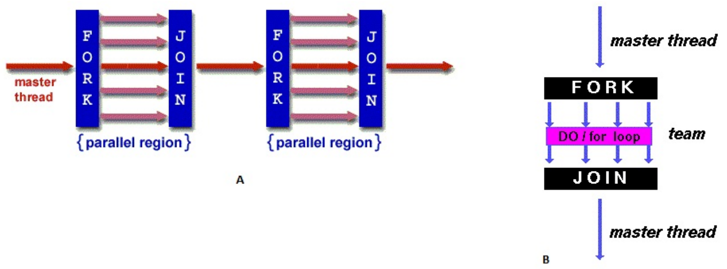


Figure 6.18: Fork Join Model, enabling mixture of sequential and parallel code with the management of a pool of threads. A : OpenMP fork and join model. B : for-loop parallelism in OpenMP. Adapted from Barney (2011).

Another very interesting fact about for-loop parallelism is the automatic barrier at the end of the loop, so all the threads wait at the end of the loop when they finish execution, thus ensuring synchronisation.

6.3.5.2.2 OpenMP Integration and Evaluation The simulation algorithm in section 5.1 clearly reveals the parallel potential of the code. It is made up of basically three sections, calculation of density, calculation of the forces and integration. Each of these tasks is performed in a similar fashion by looping through all the particles and all the iterations are completely independent from each other, thus facilitating the use of OpenMP.

Algorithm 7: Modified algorithm to perform simulation of fluid using OpenMP directives

```
while timer ticks do
    Refresh neighbour search structure
    #pragma omp for
    foreach particle P in particle list do
        Calcule the density and pressure of P
    end
    #pragma omp for
    foreach particle P in particle list do
        Calculate net force of P
        Integrate acceleration to get velocity v and position x
        Apply collision detection and resolution on P against environment
        Move P and render it
    end
end
```

The performance of the OpenMP integration was observed (figure 6.19), once again, using the two methods, *on the fly* and *big neighbour list* for neighbour search, discussed earlier in section 6.3.5.1.

The results, from figure 6.19, proved to be totally opposite of what was demonstrated in the previous graphs. Here, the *on the fly* method proved to be the fastest with performance boost of up to 3 times, while the *big neighbour list* was totally inefficient and sometimes even slower. This latter delay can be explained by the fact that the big list becomes a shared object when used with OpenMP; as such, the OpenMP *ordered* construct must be used to ensure that it is updated by only one thread at a time and that too in order. All this synchronisation among threads impaired their parallel working and slowed down the execution. As for the former case, the performance gained was because of the use of the three other processors (the simulation computer is a quad-core machine) resulting in a CPU usage of around 325 - 400 % as compared to the initial 96 - 100 %.

In this project, the **on the fly** neighbour search method was finally chosen with OpenMP integration. The graph 6.20 demonstrates the performance gain by using OpenMP.

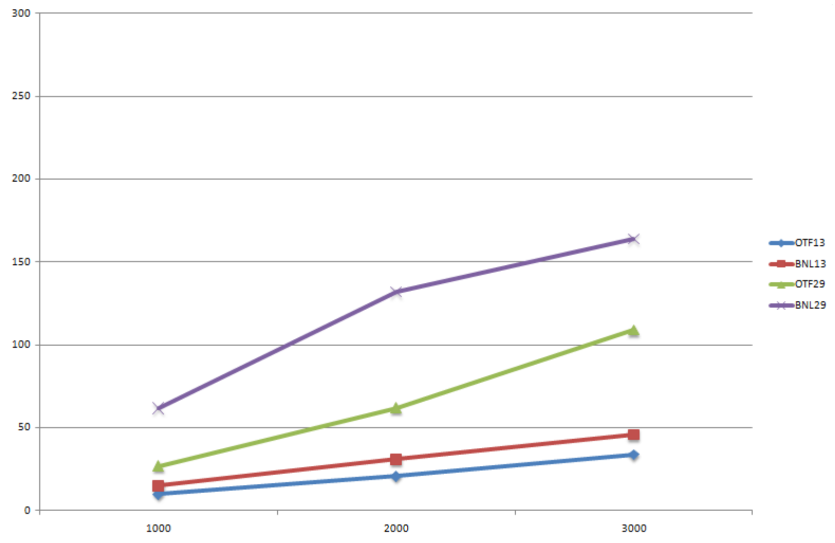


Figure 6.19: Execution time (in minutes) over iterations using OpenMP. OTF13 - on the fly with 13566 particles, OTF29 - on the fly with 29667 particles, BNL13 - big neighbour list with 13566 particles, and BNL29 - big neighbour list with 29667 particles.

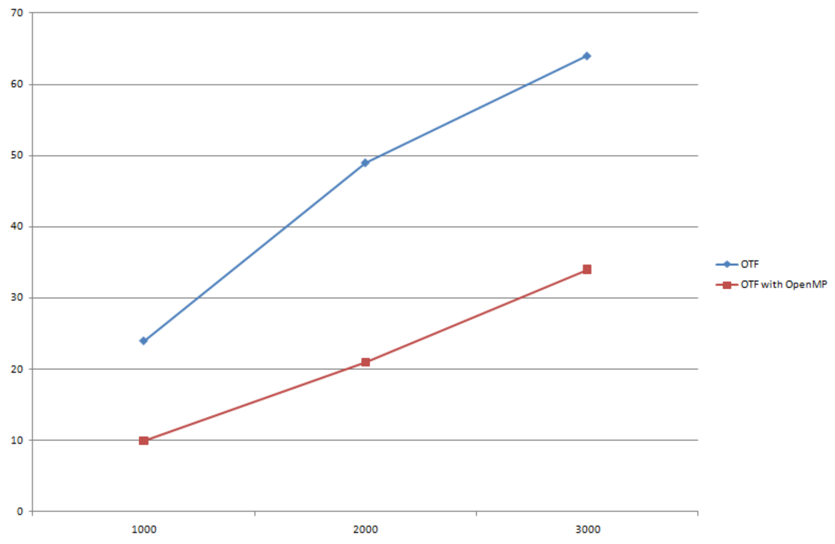


Figure 6.20: Execution time (in minutes) of 13566 particles over 3000 iterations using OpenMP and the *on the fly* method for neighbour search.

Chapter 7

Conclusion

In this project, a Lagrangian simulation of liquids has been implemented using the SPH method. In this section, an analysis of the success of the project is made, followed by a discussion of some problems encountered during the implementation and some improvements and future work.

7.1 Critical Analysis

Overall, this project has been successful as it meets all of the objectives initially set in the Design chapter 4.1. Features such as particle injection have brought in some very interesting results as shown in section 6.2.6. The interaction between the fluids and the rigid bodies has been another very successful objective achieved in this project and some very interesting scenarios are demonstrated with great success in section 6.2.3. The interaction between multiple fluids is inherent in Lagrangian particle-based methods and has proved to work really well in this project as demonstrated in section 6.2.2. The functionality of exporting the simulation data to Houdini has also been implemented and being able to manipulate it in Houdini gives much freedom for advanced enhancement and rendering. The use of multiprocessing and OpenMP has also been successfully used in this project to give a significant performance boost. Efficient use of memory and coding practices have been adopted, such as the use of passing values by reference to prevent the creation of object copies while easily pass addresses around and fast memory access achieved through contiguous memory allocation on the stack instead of resorting to pointers on the heap.

On the other hand, the simulation of the flow of various liquids present some challenges since the parameters cannot be tweaked over a wide range of values. While the basic liquid has been successfully simulated and used for most of the simulations presented in this paper, tweaking the parameters such as viscosity does not always guarantee a stable simulation, as discussed in section 6.3; as long as the timestep is kept really small, the simulation is assumed to be stable.

7.2 Problems Encountered

The density calculation of the particles is quite problematic because it depends highly on the accuracy on the neighbour search spatial algorithm and any fluctuation in the latter impacts the density value, the resulting pressure and the whole of the SPH force calculations. At several instances, the simulation would explode without any apparent reason; the cause was eventually traced back to an abnormality in the neighbour search procedure and this debugging process was quite painful. Another issue with SPH density calculation is that surface particles have less neighbours than those in the middle of the fluid; thus their densities are wrongly calculated, resulting in visual artifacts at the surface of the fluid, such as a massive concentration of particles stuck together at the edge of the boundary. Lerer, among many others, proposes solutions to solve this problem.

Another significant problem, that has been encountered, is with the neighbour search algorithm. Spatial hashing uses a smoothing length and thus divides the space in small cells; this, however, is rendered totally useless as the number of particles increases. In the latter case, the cell size remains constant and thus the number of neighbours per cell returned is massive and increases the complexity of the force calculations. At one point during the testing phase, a massive neighbour count of nearly 900 was returned and this resulted in a definite halt in the simulation, as the forces were calculated over 900 particles. To resolve this issue, the boundary dimension was increased so that the fluid particles do not come too close to each other. However, many papers have studied this problem and the most popular solution is to use adaptive smoothing length, whereby the smoothing length and thus the cell size is adjusted accordingly to the simulation space.

7.3 Improvements and Future work

While SPH is an easy method to understand and implement, it poses a lot of challenges that must be addressed to ensure that good simulations are obtained, as discussed in section 6.3. Improvements could be made to address a number of these issues. Alternate formulas, such as the Tait's equation (Monaghan, 1994), could be used to calculate the density and pressure with better accuracy. The high compressibility issues could be resolved with the use of the conjugate gradient method (Becker and Teschner, 2007). Furthermore, the timestep limit could be resolved by using implicit integration methods as discussed by Stam (1999). SPH provides a lot of room for improvement so that much more stable simulation can be achieved. Once the latter is achieved, various types of liquid can then be easily simulated such as honey and ink.

The nature of this project has a big scope for future work. Some of these are discussed briefly in this paragraph. Future work could be done on the user interface to allow easy tweaking of the simulation fluids. Concerning the representation of rigid bodies, the use of polygon meshes would be considered in the future because they provide much more complex shapes as compared to the currently being used implicit functions. This would enable much more complex and interesting scenarios such as the flow of fluid in blood vessels and etc. The performance of the simulation algorithm would be another concern for future work. Improvement, in terms of frame-rate-per-second, would be considered using other alternative technologies such as MPI to add in much more processing power or even OpenCL. Since the weakness of the spatial hashing with massive particle count has been discussed in the previous paragraph, some future work would definitely involve the use of a more efficient neighbour search algorithm designed to handle a much larger particle count, however close they get to each other, or even improve the existing spatial hashing algorithm to tackle this specific problem. The implementation of the reaction of fluid particle forces on the rigid bodies would also be a good consideration as a future work, so that Newton's third law of motion, action and reaction, is respected and thus a more stable simulation could be expected. This could also be then used to implement floating bodies on the surface of the fluid or enable more complex scenarios such as the opening of a valve by the pressure exerted by the simulated fluid.

References

Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial. *Computer*, 29:66–76. Available from: `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf` [Accessed 19 Aug 2011].

Angst, R. (2007). Fluid effects in cutting simulations. Semester thesis, Swiss Federal Institute Of Technology, Zurich. Available from: `http://www.inf.ethz.ch/personal/rangst/publications/ST07.pdf` [Accessed 19 Aug 2011].

Auer, S. (2008). Realtime particle-based fluid simulation. Master's thesis, Technical University Munich. Available from: `http://www.google.co.uk/url?sa=t&source=web&cd=2&ved=0CCIQFjAB&url=http%3A%2F%2Ffluid-particles.googlecode.com%2Ffiles%2Frealtime%2520particle%2520based%2520fluid%2520simulation%2520-%2520thesis.pdf&rct=j&q=Realtime%20particle-based%20fluid%20simulation%20auer&ei=3AxOTqiLFIeHhQf7qMDeBg&usg=AFQjCNH6qXUW7Pi-jNjsuOAOo3hOLrQYgA&sig2=cZm71gZz7Q0pTAmq-9WD-A` [Accessed 19 Aug 2011].

Barney, B. (2011). Openmp. Technical report, Lawrence Livermore National Laboratory. Available from: `https://computing.llnl.gov/tutorials/openMP/` [Accessed 19 Aug 2011].

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

Bell, N., Yu, Y., and Mucha, P. J. (2005). Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 77–86, New York, NY, USA. ACM.

Bridson, R. and Müller-Fischer, M. (2007). Fluid simulation: Siggraph 2007 course notes. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 1–81, New York, NY, USA. ACM.

Carlson, M., Mucha, P. J., Van Horn, III, R. B., and Turk, G. (2002). Melting and flowing. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '02, pages 167–174, New York, NY, USA. ACM.

Clavet, S., Beaudoin, P., and Poulin, P. (2005). Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 219–228, New York, NY, USA. ACM.

Desbrun, M. and Cani, M.-P. (1996). Smoothed Particles: A new paradigm for animating highly deformable bodies. In Boulic, R. and Hegron, G., editors, *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, pages 61–76, Poitiers, France. Springer-Verlag. Published under the name Marie-Paule Gascuel.

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). Gpu cluster for high performance computing. *SC Conference*, 0:47. Available from: `http://www.cs.sunysb.edu/~vislab/papers/GPUcluster_SC2004.pdf` [Accessed 19 Aug 2011].

Gingold, R. and Monaghan, J. (1982). Kernel estimates as a basis for general particle methods in hydrodynamics. *Journal of Computational Physics*, 46(3):429 – 453.

Harada, T., Koshizuka, S., and Kawaguchi, Y. (2007). Smoothed particle hydrodynamics on gpus. In *Proc. of Computer Graphics International*, pages 63–70.

Harris, M. J. (2003). *Real-time cloud simulation and rendering*. PhD thesis. Available from: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.7402&rep=rep1&type=pdf` [Accessed 19 Aug 2011].

Hoetzlein, R. and Höllerer, T. (2009). Interactive water streams with sphere scan conversion. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 107–114, New York, NY, USA. ACM.

Horvath, P. and Illes, D. (2007). Sph-based fluid simulation for special effects. In *The 11th Central European Seminar on Computer Graphics*. Available from: `http://www.cg.tuwien.ac.at/hostings/cescg/CESCG-2007/papers/TUBudapest-Horvath-Peter/TUBudapest-Horvath-Peter.pdf` [Accessed 19 Aug 2011].

Hut, P. and Makino, J. (2004). The art of computational science. Available from: `http://www.artcompsci.org/kali/vol/two_body_problem_2/ch02.html#rdocsect8` [Accessed 19 Aug 2011].

Kelager, M. (2006). Lagrangian fluid dynamics using smoothed particle hydrodynamics. Available from: `http://image.diku.dk/projects/media/kelager.06.pdf` [Accessed 19 Aug 2011].

Kolb, A. and Cuntz, N. (2005). Dynamic particle coupling for gpu-based fluid simulation. In *In Proc. of the 18th Symposium on Simulation Technique*, pages 722–727.

Lerer, A. 3-dimensional fluid simulation with smoothed particle hydrodynamics. Available from: `http://web.mit.edu/alerer/www/sph.pdf` [Accessed 19 Aug 2011].

Lucy, L. (1977). A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024.

Macey, J. (2010). Ngl graphics library. Available from: `http://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/` [Accessed 19 Aug 2011].

Miller, G. and Pearce, A. (1989). Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics*, 13(3):305 – 309.

Monaghan, J. J. (1994). Simulating free surface flows with sph. *Journal of Computational Physics*, 110(2):399 – 406.

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

Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association.

Müller, M., Solenthaler, B., Keiser, R., and Gross, M. (2005). Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 237–244. ACM.

Paiva, A., Petronetto, F., Lewiner, T., and Tavares, G. (2009). Particle-based viscoplastic fluid/solid simulation. *Computer-Aided Design*, 41(4):306–314.

Pelfrey, B. (2010). An informal tutorial on smoothed particle hydrodynamics for interactive simulation. Available from: `http://www.cs.clemson.edu/~bpelfre/sph_tutorial.pdf` [Accessed 19 Aug 2011].

Premoze, S., Tasdizen, T., Bigler, J., Lefohn, A., and Whitaker, R. T. (2003). Particle-based simulation of fluids. *Computer Graphics Forum*, 22(3):401–410. Available from: `http://www.sci.utah.edu/~tolga/pubs/ParticleFluidsHiRes.pdf` [Accessed 19 Aug 2011].

Priscott, C. (2010). 3d langrangian fluid solver using sph approximations. Master's thesis, Bournemouth University, NCCA. Available from: `http://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc2010/08ChrisPriscott/ChrisPriscott_THESIS.pdf` [Accessed 19 Aug 2011].

Reeves, W. T. (1983). Particle systems : a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2:91–108.

Stam, J. (1999). Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co. Available from: `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf` [Accessed 19 Aug 2011].

Stam, J. and Fiume, E. (1995). Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on*

Computer graphics and interactive techniques, SIGGRAPH '95, pages 129–136, New York, NY, USA. ACM.

Steele, K., Cline, D., Egbert, P. K., and Dinerstein, J. (2004). Modeling and rendering viscous liquids: Research articles. *Comput. Animat. Virtual Worlds*, 15:183–192.

Succi, S., Benzi, R., and Higuera, F. (1991). The lattice boltzmann equation: A new tool for computational fluid-dynamics. *Physica D: Nonlinear Phenomena*, 47(1-2):219 – 230.

Takeshita, D., Ota, S., Tamura, M., Fujimoto, T., Muraoka, K., and Chiba, N. (2003). Particle-based visual simulation of explosive flames. *Computer Graphics and Applications, Pacific Conference on*, 0:482.

Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., and Gross, M. (2003). Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV03*, pages 47–54. Available from: `http://www.beosil.com/download/CollisionDetectionHashing_VMV03.pdf` [Accessed 19 Aug 2011].

Umenhoffer, T. and Szirmay-Kalos, L. (2008). Interactive distributed fluid simulation on the gpu. Technical report, Budapest University of Technology and Economics. Available from: `http://sirkan.iit.bme.hu/~szirmay/fluid.pdf` [Accessed 19 Aug 2011].

WikimediaCommons (2011). Sine(sin)-function. Available from: `http://commons.wikimedia.org/wiki/Image:Sine.svg` [Accessed 19 Aug 2011].