Fluid Simulation using Smooth Particle Hydrodynamics

Jennifer Moorehead

August 22, 2016

MSc Computer Animation and Visual Effects

Contents

Abstract

This paper discusses the implementation of a fluid solver which implements a version of Smooth Particle Hydrodynamics aproximations using the prediction relaxation scheme proposed by Clavet et al. In addition optimisations such as spacial hashing have been made and the point geometry has been exported to Houdini for meshing, lighing and rendering purposes. Examples of the project can be found in the Results section.

1 Introduction

The outline of this paper follows the process of design and implementation of a 3D Lagrangian approach to fluid simulation. This solver utilises a particle system and SPH smoothing kernels to solve the incompressible navier stokes equations for fluid dynamics. Each particle holds its own fluid attributes such as density and pressure. These terms can be evaluated over time and a prediction relaxation scheme is used to produce new positions for each particle. In addition a fast nearest neighbour searching technique known as spacial hashing has been implemented to provide optimisations to one of the slowest components of a Lagrangian solver. Boundary conditions have been put in place and can be updated to produce a rolling wave effect.

The following sections discuss the previous work in particle fluid simulation, and explanation of navier stokes for particle fluid solvers. Following that the design and implementation of the solver is outlined in detail and the results are shown. Please refer to the accompanying videos for better visualisation of the results.

2 Previous Work

Smoothed Particle Hydrodynamics is a relatively young approach to fluid simulation. It was initially proposed as a method for simulating nonaxisymmetric phenomena in astrophysics by Monaghan and discovered to give sensible solutions to difficult situations [8]. He introduced the core of SPH through the approximation formula, which can be used to solve the incompressible navier stokes equations for fluid motion. This ground work was then expanded on by Desbrun and Gascuel [2] specifically to animate highly deformable bodies. This paper shows that the SPH paradigm can animate inelastic bodies with properties such as stiffness and viscosity. It also introduces a variation of the ideal gas state equation which is still a popular approach to solving the pressure term for fluid within Smooth Particle Hydrodynamics. Numerous papers followed from these foundations, and allowed researchers to study the use of SPH to simulate highly viscous fluids. For example Clavet et al [12]achieved realistic behaviour of viscoelastic substances on a small scale, through elastic and plastic springs with varying rest length. Concurrent with these areas, Muller proposed a real time approach to SPH for interactive application [6] and Solenthaler developed an approach which forces fluid incompressibility through a prediction correction scheme to determine particle pressure [13]. Areas of interest within SPH have also been advanced through research into new smoothing kernel functions, and improvements to particularly slow components of the system. For example Teschner et al developed a fast nearest neighbour searching algorithm which utilises spacial hashing functions [14], continuing to improve the SPH approximation paradigm for a relatively simlpe approach to fluid simulation.

3 Navier Stokes and Fluid Theory

3.1 Lagrangian Approach

Named after the French mathematician Lagrange, this is a more commonly understood method of tracking motion and the approach taken for this project. It treats the fluid as a particle system. This means that the fluid is made up of particles, each with its own fluid properties such as a position $\overrightarrow{x}$ and a velocity $\overrightarrow{u}$. Each particle could be thought of as a single molecule of fluid reacting to the fluid quantities (pressure, density) around it[1]. This method is not spacially limited, however a large number of particles is necessary for realistic simulation, which decreases performance.

3.2 Eularian Approach

Named after the Swiss mathematician Euler, this is the classical technique used for fluid simulation. It tracks fluid quanties such as density at fixed points in space. The motion is described by the change in a velocity field, a density field and a pressure field over time [6]. The fluid flows past these points and contributes some sort of change. For example if a cold fluid moves past a fixed point, followed by a warm fluid, the temperature at this point will begin to increase. Unlike the Lagrangian approach, the Eularian method is not hindered by the number of particles used, making it a fast alternative. However it is limited by the resolution of the grid used to represent the fields.

3.3 Navier Stokes

The motion of Fluid flow is governed by a set of partial differential equations known as the Incompressible Navier Stokes equations.

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

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

3.3.1 The Incompressibility Equation

The first equation 3.1 is known as the Incompressibility equation. In Eularian based methods this insures that mass is conserved throughout the fluid. However, using a particle based approach allows this equation to be omitted entirely. Since this approach uses a constant number of particles, each having a fixed mass, conservation of mass is guaranteed [6].

3.3.2 The momentum Equation

The second equation 3.2 is known as the Momentum Equation. It informs how the the fluid moves due to the forces that act upon it. More simlpy put, this

is Newtons Second Law of Motion $F = ma$, which states that momentum is always conserved [7]. If Newtons Second Law is rearranged it gives $a = \frac{F}{m}$. Finding the acceleration of the fluid is necessary in order to integrate and find a new position for each particle. In the Eularian approach, advection is given by the term $\frac{\partial u}{\partial t} + v \cdot \nabla v$. However, since the Lagrangian particles contain the fluid quantities, this can be replaced by substancial derivitive $\frac{Dv}{Dt}$ of the velocity of the particles [3]. So the momentum equation can be simplified to:

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

The simplest force acting upon the fluid is the external force due to gravity, represented in equation 3.3 by the letter g. This is generally taken to be $(0, -9.8, 0) m/s^2$.

$$-\nabla p \tag{3.4}$$

The first of the fluid forces is pressure. Differences in pressure on one side of the fluid particle result in fluid flow towards the side of lower pressure along the negative gradient for pressure $-\nabla p$. Pressure attempts to balance the density differences thoughout the fluid. (Density is mass per unit volume).

$$\mu \nabla^2 v \tag{3.5}$$

The other internal force contributing to the fluid motion is viscosity. A viscous fluid tries to resist deformation, Fluids like molasses have high viscosity, while fluids like mercury have low viscosity. It measures how difficult the fluid is to stir [1]. Details on how to derive pressure and viscosity are outlined in section 5. The sum of these forces determine the change of momentum. Therefore the momentum equation 3.2 for particle i can be simplifed to the following:

$$a_i = \frac{F_i}{m_i} \tag{3.6}$$

$$F = \mathbf{F}pressure + \mathbf{F}viscosity + g \tag{3.7}$$

$$a_i = \frac{Dv_i}{Dt} = \frac{F_i}{\rho_i} \tag{3.8}$$

3.4 SPH Theory

Smoothed Particle Hydrodynamics is a method for obtaining approximate numerical solutions for the equations of fluid dynamics by replacing the fluid with a set of particles [9]. Equation 3.2 becomes an ordinary differential equation, and SPH can be used to solve for each fluid quantity [8]. It uses the discretised particle locations to evaluate the fluid quantity using the weighted sum of neighbouring particles [10].

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

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

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

Where $A(x)$ is a fluid quantity at position x that needs to be solved. j iterates through all the particles, m_j is the mass of particle j, ρ_j the density, and A_j is the fluid quantity at position j. $W(x - x_j, h)$ is a radially symmetric smoothing kernel with a radius h. This weights the contribution of the quantity at position j according to its distance from position x. Particles which lie beyond the radius h have no contribution to the interpolation of the quantity $A(x)$. Pressure and Viscosity can be calculated relatively easily using SPH techniques , however the resultant forces can be asymmetric. It is important to remember that SPH has inherent problems. These equations are not guaranteed to satisfy symmetric forces and conservation of momentum, and can therefore introduce instabilities into the system. The smoothing kernels used are discussed in section 5

4 Design

4.1 Class Diagram

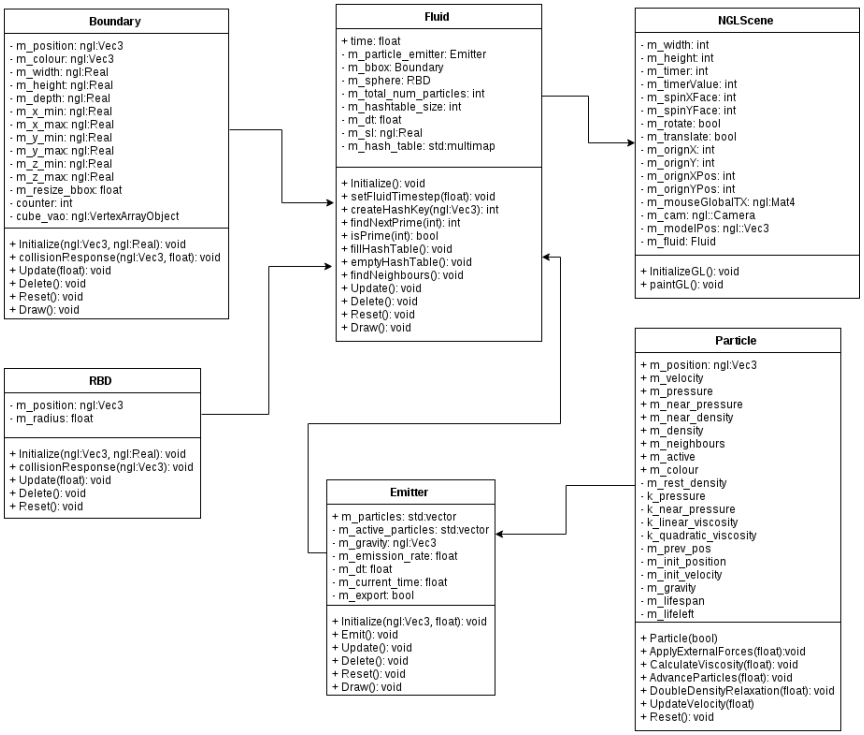

Figure 4.1: Fluid Solver UML Diagram

Figure 4.1 dipicts how the classes of the fluid solver interact with each other.

4.1.1 Fluid Class

This class provides the set up for the solver. This is where the particle system is created and gets updated. Each of the SPH smoothing methods is called on each particle from the update of this class. It also houses the methods which facilitate nearest neighbour searching for each particle.

4.1.2 Particle Class

This class represents a single particle of the fluid body. It houses the smoothing kernel functions to solve the SPH terms, and methods to advance the particle's position.

4.1.3 Emitter Class

This class represents the particle system of the fluid body. It initializes a user defined number of particles that the fluid should consist of. The particles have a flag which if initially set to false, will allow the Emit() function to emit a certain user defined amount of particles at a user defined emission rate. If this rate is set to 0 then there will be a constant emission, and the fluid will appear as if from a faucet.

4.1.4 Boundary and RBD Classes

These classes represent the tank and the sphere collision objects respectively. They house each objects collision detection and response functions and the boundary object contains an update method which will allow the user to simulate a rolling wave of fluid.

5 Implementation

5.1 Fast Neighbour Searching

A Lagrangian approach to fluid simulation requires iterating through each particle in turn to solve each component of the navier stokes equation (pressure, viscosity). This means that the time complexity for a Lagrangian fluid with n particles checking against every other particle is $O(n^2)$.According to SPH interpolation methods, to find the fluid terms each particle i must iterate through a list of neighbour particles j within a search radius h. As the number of particles increases, a naive time complexity becomes less and less useable, and since tens of thousands of particles are necessary for realistic simulation, optimisations have to be made.

5.1.1 Spacial Hashing

This optimisation technique falls under the NNS(Nearest Neighbour Search) umbrella. This is a method of splitting the world space into a grid of keys or "cells". Each particle is hashed by its 3D position to a 1D hash table [14]. The idea behind this is that the hash function should assign particle positions which are close to each other to the same hash key. In theory this means that each particle need only know its own hash key to find its list of neighbours. This should provide a performance increase to $O(1)$. However this method is dependant upon a hash function which provides unique keys and how fast they can be generated [4]. Therefore Teschner et al propose the following function:

$$hash(\hat{x}) = (\hat{x}_x p_1 \textbf{ xor } \hat{x}_y p_2 \textbf{ xor } \hat{x}_z p_3) mod\ n_H \tag{5.1}$$

Where $\hat{x}$is a function which discretizes a 3D point or particle position.

$$\hat{x}(x) = ([x_x/l], [x_y/l], [x_z/l]) \tag{5.2}$$

This function takes in a floating point vector representing a 3D particle position, and creates a new integer vector based on a defined cell size l. This cell size is equal to the radius h used in the SPH smoothing kernels discussed in section 5.2. p_1,$p_2$and $p_3$are large prime numbers, this implementation uses the same found in Teschner et al [15].

$$p_1 = 73,856,093 \tag{5.3}$$

$$p_2 = 19,349,663 \tag{5.4}$$

$$p_3 = 83,492,791 \tag{5.5}$$

The final unknown in equation 5.1 is n_H. This is the size of the hash table, which significantly influences the performance of the algorithm. If the hash table is sufficiently larger than the number of particles, this reduces the possibility of hash key collisions. The hash function works most efficiently if the hash table size is a prime number [14]. The size of the hash table can be computed with the following:

$$n_H = prime(2n) \tag{5.6}$$

Where n is the number of particles, and $prime(x)$is a function which returns the next prime number after x [11]. Before the neighbours for any particular particle can be found the hash table needs to be filled.

$$hashtable[hashkey(\hat{x}(x_i))] = Particle_i \tag{5.7}$$

This implementation uses the C++ Standard Template Library's multimap data structure to represent the hash table. This is due to its ability to assign multiple values to the same key, which is ideal for our circumstances. Once the hash table has been appropriately filled, then neighbours can be found using a particle's hash key. However, there is no guarantee that all of the particles with the same hash key as the particle in question will actually be neighbours of that particle. To account for this, two corners of a bounding box are used to filter out potential neighbours.

$$BBmin = \hat{x}(x_q - (h,h,h)^2),\ BBmax = \hat{x}(x_q + (h,h,h)^2) \tag{5.8}$$

Equation 5.8 represents the two corner points of this bounding box. x_q is the particle that is being queried and if it lies somewhere within the bounding box then this particle is added to the neighbour list. See algorithm 1.

Algorithm 1 Nearest Neighbour Searching using Spacial Hashing

```
foreach particle i
//create a key and fill the hash table
    hash_key←createHashKey(x̂(x_i))
    hash_table.insert(hash_key, &i)
    smooth_length← (h, h, h)^2
//find neighbours
foreach particle i
    i.neighbours.clear()
    bmin ← x_i−smooth_length
    bmin ← x_i+smooth_length
    for x = bmin_x; x < bmax_x; x+ = h
        for y = bmin_y; y < bmax_y; y+ = h
            for z = bmin_z; z < bmax_z; z+ = h
                x_q ← (x, y, z)
                //create a key to test based on the bounding box
                test_key = createHashKey(x̂(x_q))
                //query the hash table
                hash_table.equal_range(test_key)
                foreach particle j in test_key
                    inlist = false
                    for k = 0k < i.neighbours.size()k + +
                        if j = i.neighbours[k]
                            inlist = true
                            break;
                    if inlist = false
                    d = i(x, y, z) − j(x, y, z)
                    if | d |<smooth_length
                        x.neighbours.push_back(j)
```

5.2 Double Density Relaxation Smoothing Kernels

Initially the kernels used where those proposed by Muller et al [6]. The poly6 kernel for calculating density, and Debrun's spiky kernel for calculating pressure. However, satisfactory results were not achieved with this method, therefore an approach proposed by Clavet et al was utilised [12]

5.2.1 Calculating Density

The Density for particle i is approximated by summing the weighted contributions of each neighbouring particle j using the following quadratic spike kernel:

$$\rho_i = \sum_{j \epsilon N(i)} (1 - r_{ij}/h)^2 \tag{5.9}$$

Where $r_{ij} =\mid r_{ij} \mid, r_{ij} = x_j - x_i$. x_i is the position of the particle currently being

evaluated, and x_j is the position of one of its neighbouring particles. $N(i)$ denotes the list of neighbours for particle i. It should be noted that this is not a true physical property for density, but provides a number which quantifies how the particles which make up the fluid relate to their neighouring particles [12].

5.2.2 Calculating Pressure

This force corresponds to the difference between the current density ρ_i and a defined rest density ρ_0. Rest density is the ideal density that the fluid will try to maintain. The pseudo-pressure is calculated for each particle using Desbrun's variation of the ideal gas state equation [2].

$$P = k\rho \tag{5.10}$$

$$P_i = k(\rho_i - \rho_0) \tag{5.11}$$

Where k is a constant parameter controlling stiffness. If the current density ρ_i is greater than the rest density, pressure is a repulsive force, if ρ_i is less than the rest density, then pressure is an attractive force. The density relaxation displacement between two particles is proportional to the pseudo pressure force, which is weighted by the following linear kernel function:

$$D_{ij} = \triangle t^2 P_i(1 - r_{ij}/h)\hat{r} \tag{5.12}$$

Where $\hat{r}_{ij}$ is the unit vector from particle i to particle j. The displacement is applied directly to particle j and an equal and opposite displacement is applied directly to particle i. The time step is squared to account for the double integration of the force to get both displacement values.

5.2.3 Calculating Near Density

Near density and pressure are calculated in order to prevent particle clusters. Sometimes particles reach the rest density by attracting its neighbours. This results the fluid separating into multiple clusters, rather than one continuum . To solve this issue an additional pressure term is added. To find this new pressure term, near density is calculated with a cubic spike kernal function.

$$\rho_i^{near} = \sum_{j \epsilon N(i)} (1 - r_{ij}/h)^3 \tag{5.13}$$

This results in a sharper spike than the standard density kernel 5.9

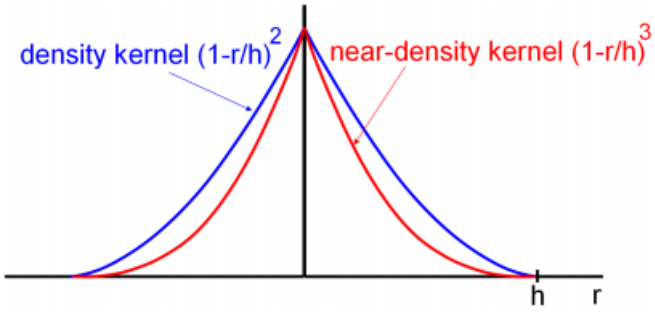

Figure 5.1: Density and Near Density Kernel Comparison [12]

5.2.4 Calculating Near Pressure

To account for particle clustering, the near pressure force must only give repulsive values. Therefore rest density is removed from equation 5.11.

$$P_i^{near} = k^{near}\rho^{near} \tag{5.14}$$

Like pressure, near pressure gives equal and opposite displacement forces for two particles, this is the second step of the double density relaxation. Therefore equation 5.12 is adjusted.

$$D_{ij} = \Delta t^2 \left(P_i(1 - r_{ij}/h) + P_i^{near}(1 - r_{ij}/h)^2\right)\hat{r}_{ij} \tag{5.15}$$

The quadratic spike kernel defines how near repulsion force is applied to the neighbouring particles [12]. See Algorithm 2

Algorithm 2 Double Density Relaxation

foreach particle i
$\rho \leftarrow 0$
$\rho^{near} \leftarrow 0$
//compute density and near density
foreach particle j ϵ neighbours(i)
    $q \leftarrow r_{ij}/h$
    if $q < 1$
        $\rho \leftarrow \rho + (1-q)^2$
        $\rho^{near} \leftarrow \rho^{near} + (1-q)^3$
//compute pressure and near pressure
$P \leftarrow k(\rho - \rho_0)$
$p^{near} \leftarrow k^{near}\rho^{near}$
$\boldsymbol{dx} \leftarrow 0$
foreach particle j neighbours(i)
    $q \leftarrow r_{ij}/h$
    if $q < 1$
        //apply displacements
        $\boldsymbol{D} \leftarrow \Delta t^2(P(1-q) + P^{near}(1-q)^2)\hat{r}_{ij}$
        $\boldsymbol{x}_j \leftarrow \boldsymbol{x}_j + \boldsymbol{D}/2$
        $\boldsymbol{dx} \leftarrow \boldsymbol{dx} - \boldsymbol{D}/2$
$x_i \leftarrow x_i + \boldsymbol{dx}$

5.2.5 Calculating Viscosity

The paper this implementation is based on [12] uses viscoelastic behavior, which is introduced through elastic and plastic spring displacements as well as viscosity. However our implementation omits the springs and uses instead only the linear and quadratic formula. Viscosity is another smoothing kernel function which smooths the velocity between neighbouring particles. The viscosity function adjusts the particle velocity at the beginning of each timestep. See Algorithm 3

Algorithm 3 Viscosity

foreach neighbour pairij, $(i < j)$
    $q \leftarrow r_{ij}/h$
    if $q < 1$
        //inward radial velocity
        $u \leftarrow (v_i - v_j)\cdot \hat{r_{ij}}$
        if $u > 0$
            linear and quadratic forces
            $\mathbf{I} \leftarrow \Delta t(1-q)(\sigma u + \beta u^2)\hat{r_{ij}}$
            $v_i \leftarrow v_i - \mathbf{I/2}$
            $v_j \leftarrow v_j + \mathbf{I}/2$

The difference between particle j's velocity and particle i's velocity $(v_i - v_j)\cdot \hat{r_{ij}}$

measures the speed at which particle j is travelling towards particle i. The application of the force is dependant upon the distance between the particles. This is achieved through the use of the linear kernel function $(1-q)$ where $q = r_{ij}/h$. $(\sigma u + \beta u^2)$ controls the linear viscosity and quadratic viscosity respectively. To achieve a very viscous fluid the linear viscosity σ should be increased. Quadratic viscosity prevents particle to particle penetration by removing very high inward velocities [12]. It should be noted that for low/non viscous fluid such as water, the purpose of viscosity is to handle particle to particle collision. Therefore viscosity should only be applied if the particle tragectories are on target for a collision.

5.3 Integration

The integration approach taken is similar to an involved implicit scheme, but is much faster and simplier than such methods. Rather than computing the forces and updating the velocities for each particle to find a new position, particles are moved according to their velocities, and then relaxed by the double density smoothing kernels.

Algorithm 4 Integration

foreach particle i
$x_i^{-1} \leftarrow x_i$
$x_i \leftarrow x_i + v_i \Delta t$

By using forces that exist further ahead in time, instabilities are predicted before they are introduced into the system and can be accounted for.

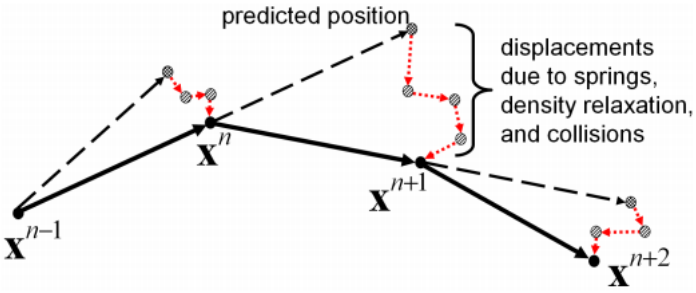

Figure 5.2: Integration

5.4 Collision Detection

In order to have an interesting simulation, the fluid object should collide with other objects in the scene. Without collision with other objects, the fluid motion would be rather difficult to see in action. Therefore collision within a tank and collision with spherical rigid bodies have been included in this implementation.

5.4.1 Tank Boundary Objects

Being unbound by a fixed grid is a significant advantage of a Lagrangian approach to fluid simulation. However it is necessary to give the fluid restrictions to reside inside to display the fluid motion. This meant the inclusion of a boundary object for the particles to collide and interact with. Detection is implemented very simply. The boundary is given a minimum and maximum position in the x, y and z directions. Algorithm 5 shows collision detection and response with the right most x position of the bounding box.

Algorithm 5 Collision Detection with the Boundary

foreach particle i
if $i_x > bbox_x$
    $i_x \leftarrow bbox_x$
    $v_x \leftarrow v_x * d$

Where v_x is the velocity in the x direction of particle i and d is a negative damping constant between 0 and 1. Damping gives the impression that the particle loses energy upon impact with the boundary. As can be seen, if the particle's x position is greater than the maximum x position, then the particle's x position is set to the boundary with which it collided, and its velocity is reversed and dampened.

5.4.2 Updating the Boundary

In order to achieve a rolling wave demonstration, an update was implemented for the boundary object. Within this method if the time is greater than a user specified time, the right most x position of the bounding box will be decreased at a constant rate using a user defined parameter. Once the right most x position reaches the ideal user defined minimum position, this parameter is inverted to increase the x position to its original length. A sine wave function was considered for this implementation, but this method gives preferable control over the movement.

5.4.3 Spherical Collisions

To add further visual interest and complexity collisions with spherical rigid bodies where implemented, both as a collision object and a container. A sphere is defined implicitly by a center point (x, y, z) and a radius r.

Algorithm 6 Collision with the Outside of a Sphere

foreach particle i
$d \leftarrow x_i - x_s$
if $\mid d \mid < r^2$
    $n \leftarrow d$
    $x1 = \hat{n} \cdot v_i$
    $v1x = \hat{n}x1$
    $v1y = v_i - v1x$
    $x_i \leftarrow x_i - \hat{n}(r - \mid d \mid)$
    $v_i \leftarrow (-v1x + v1y) * k$

Where $\hat{n}$ is the normalised normal of d and k is a damping force. If the length of the distance vector between the particle i and the centre point of the sphere x_s is greater than the radius squared then a collision has occurred. To correct the dot product of normal $\hat{n}$ and the velocity of the particle v_i is used to create two new vectors $v1x$ and $v1y$. The position of the particle i is then adjusted along the normal, and the velocity v_i is updated using the two new vectors and a damper to mimic the loss of energy upon impact. To allow for collisions this formula is adjusted slightly.

Algorithm 7 Collision with the Inside of a Sphere

foreach particle i
$d \leftarrow x_i - x_s$
if $\mid d \mid < r^2 \&\& \ i_y < (r - p)$
    $n \leftarrow d$
    $x1 = \hat{n} \cdot v_i$
    $v1x = \hat{n}x1$
    $v1y = v_i - v1x$
    $x_i \leftarrow x_i - \hat{n}(\mid d \mid -r)$
    $v_i \leftarrow (-v1x + v1y) * k$

To determine collisions with the inside of the sphere, an additional check is made so see if the y position of particle i is less than $(r - p)$. Where r is the radius of the sphere, and p is a value less than the radius of the sphere. This parameter will allow the user to specify at which height within the sphere collisions should begin being detected. The position correction for particle i is also inverted.

5.4.4 Simulation Step

The following Algorithm outlines a single step in the simulation.

Algorithm 8 Simulation Step

```
particle_emitter.update()
foreach particle i
    if(i.active)
        i.ApplyExternalForces()
foreach particle i
    if(i.active)
        i.CalculateViscosity()
foreach particle i
    if(i.active)
        i.AdvanceParticles()
emptyHashTable()
fillHashTable()
findNeighbours()
foreach particle i
    if(i.active)
        i.DoubleDensityRelaxation()
foreach particle i
    if(i.active)
        i.UpdateVelocity()
foreach particle i
    if(i.active)
        i.checkForCollisions()
```

It should be noted that in addition to filling the hash table during the simulation update, it should be filled when the fluid is initialized.

6 Results

This section deals with the results of the SPH implementation and covers how the final looks where achieved.

6.1 Example Simulations

Below are some of the varying results which can be achieved by this fluid solver. Each demonstration's particle position information was exported from the C++ solver into Houdini using the Alembic framework [5]. Lighting and Shading where achieved in Houdini and each demo was rendered using Mantra.

Figure 6.1: Wine Fluid Results

Figure 6.1 displays a non viscous wine fluid interacting with the inside of the sphere acting as the container. Once the point geometry was imported into Houdini it needed to be converted and represented as a particle fluid surface mesh rather than simple point geometry. A simple studio background was chosen and the object is lit with three point lighting and an envirnoment map was utilised to facilitate reflections in the glass and liquid.

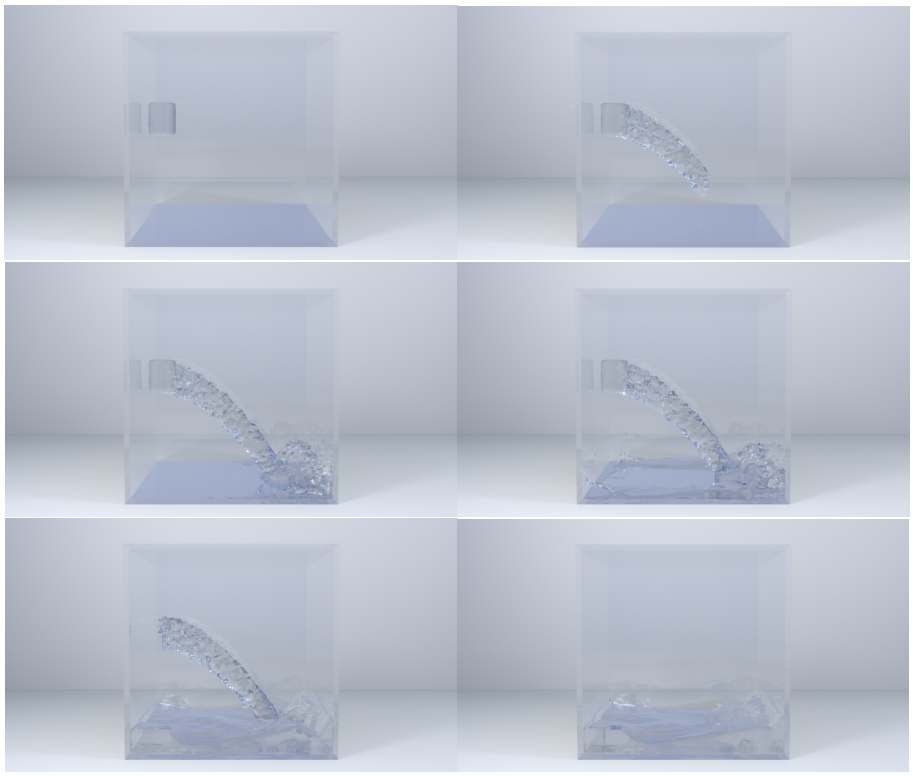

Figure 6.2: Water in a Glass Tank

Figure 6.2 shows a non viscous water fluid interacting with a cuboid boundary object acting as a container. When the point geometry was imported into Houdini it was necessary to convert the alembic object to apply a particle fluid surface. The parameters of the particle fluid surface contributed to the realism of the fluid. But in addition to this, it should be noted that to achieve satisfactory results, this demo required the solver to simulate 100,000 particles. As such the speed of the simulation decreased, as well as the export. The alembic file size also increased with the addition of particles. However additional particles yeilded better looking results from the particle fluid surface, as parameters like voxel scale could be decreased. This improved jittering artifacts that where observed on exported simulations with fewer particles. By exporting 100,000 particles this enabled stress testing the solver. Although the simulation times did increase, it is worth noting that increasing the particle count did not cause the system to become unstable. Instabilities appeared in the system when the rest density and pressure constants were set too high. These parameters require balancing. However it is possible to achieve pleasing results.

7 Conclusion

The purpose of this project was to implement a functional 3D Lagrangian fluid solver for the simulation of water. Ideally the solver would be stable and robust with an appropriate performance dependant upon the particle count. It was also an aim of the project to produce a clean and aesthetically pleasing demo. This project has been a success in most of these regards. The solver can run in real time with up to 5,000 particles, and remains stable with up to 100,000.

8 Known Issues

8.1 Gridlines

When the fluid is observed as particles, equally spaced gridlines can be seen around the fluid edges. This is minimised by finding a more balanced smoothing length for the particle count in question. However this cannot be considered a permenant fix.

8.2 Sticky Particles

In some cases, when the particles collide with their container, particularly the spherical container, the particles tend to stick to the surface of the collision object. This occurance is not due to the viscosity of the fluid but rather a non physical collision detection scheme.

8.3 Flickering

When the particle positions are imported into Houdini and a particle fluid surface is applied, sometimes flickering can be observed at the base of the fluid surface. It is suspected that this issue is linked to the spaced line grids at the fluid edge.

9 Future Work

If this project were to continue, the following would be desirable features that would be included.

9.1 Houdini Digital Asset

At present, importing the particle position information to Houdini is a manual process. Ideally this would be automated, and Houdini would act as a front end interface to the C++ solver. The asset would allow the user to specify particle counts, placement of the particle emitter and the inital positions of the fluid. The user would also be able to tweak the parameters of the particle fluid surface from within one houdini parameter interface.

9.2 Shaped Initial Positions

At present the particle fluid can be initialised to start in the shape of a cuboid, or be emitted as if from a faucet. A desirable feature would be to allow the user to specify some arbitary geometry which would act as a guide to the starting positions.

9.3 Improvecd Collision Handling

At present the collision handling in the fluid system is not phsyically based. This could be improved by using a more physically accurate collision detection and response model. An attempt was made to implement Bridson's ghost particle method to collision detection. This method initialises a set of ghost particles at the surface of the collision object, the density and pressure terms for the fluid calculations mean that the particles cannot intersect with the surface of the object. However artifacts where observed in the process of implementing this improvement and it was thought best to adhear to another more pressing issue. This method could be implemented to full functionality as an alternative collision detection and response method.

References

[1] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters, Ltd., may 2008.

[2] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. *In Computer Animation and Simulation 96 (Proceedings of EG Workshop on Animation and Simulation)*, pages 61–76, 1996.

[3] Peter Horvath and David Illes. Sph-based fluid simulation for special effects. 2007.

[4] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Master's thesis, University of Copenhagen, jan 2006.

[5] Jon Macey. Simulation exports. may 2016.

[6] David Charypar Matthias Muller and Markus Gross. Particle-based fluid simulation for interactive applications. *Eurographics/SIGGRAPH*, 2003.

[7] Nick Foster Dimitri Metaxas. Realistic animation of liquids. 1996.

[8] J J Monaghan. Smooth particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, pages 543–574, 1992.

[9] J J Monaghan. Smooth particle hydrodynamics. 2005.

[10] Rajiv Perseedoss. Lagrangian liquid simulation using sph. Master's thesis, Bournemouth University, aug 2011.

[11] Christopher Priscott. 3d langrangian fluid solver using sph approximations. Master's thesis, Bournemouth University, aug 2010.

[12] Philippe Beaudoin Simon Clavet and Pierre Poulin. Particle-based viscoelastic fluid simulation. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2005.

[13] Barbara Solenthaler and Renato Pajarola. Predictive-corrective incompressible sph. In *ACM transactions on graphics (TOG)*, volume 28, page 40. ACM, 2009.

[14] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H Gross. Optimized spatial hashing for collision detection of deformable objects. In *VMV*, volume 3, pages 47–54, 2003.

[15] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, M-P Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Strasser, et al. Collision detection for deformable objects. In *Computer graphics forum*, volume 24, pages 61–81. Wiley Online Library, 2005.