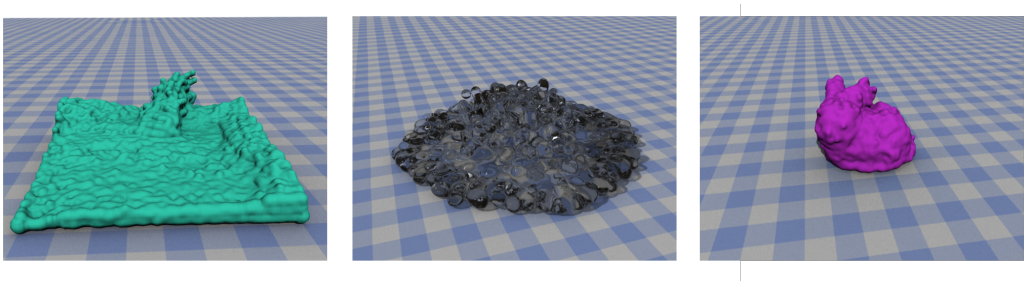
MSc Masters Project
3D Langrangian Fluid Solver using SPH approximations.

Chris Priscott
MSc CAVE 09-10

August 17, 2010



Contents

Abstract

This paper looks at the implementation of a 3D Langrangian Fluid Solver, using a collection of SPH techniques and algorithms, with full integration into the Houdini software package. New features such as temperature diffusion and optimisation methods (including spatial hashing, particle velocity correction and artificial viscosity) improve on Houdini's existing particle fluid solver allowing for a wide range of effects to be achieved through an easy to use interface. A new technique is also proposed that adjusts the pressure and density fields to create more vigourous and spectacular crashing waves. A variety of examples prove the success of this project and possible future work is suggested.

1 Introduction

The outline of this paper follows the work surrounding the design and implementation of a 3D Langrangian Fluid Solver. Using this solver, fluids are modeled using particles that each individually hold attributes such as there pressure and viscosity. A Langrangian form of the Navier Stokes equations can be used to calculate these attributes over time, allowing for there new acceleration, and hence position, to be found. Multiple 3D packages contain existing particle solvers but the one presented in this paper adds extra features from new research, and also utilises a new method to calculate the pressure force.

In the following sections there is discussion on the previous work so far with particle fluids and theory on how particle fluids and the Navier Stokes equations work. From this follows a section on the design of a suitable pipeline and program structure and a look into how this solver and pipeline is implemented. This is followed up by a analysis of the results achieved, including a review on the efficiency and known issues of the solver, and then conclusion of the relative success of the project.

2 Previous Work

Smoothed Particle Hydrodynamic's is a relatively new method that has gained a lot more interest in recent years as an alternative to Voxel Fluids. The first use was by Monaghan (1992) where he uses the method to simulate and study astronomical events such as gas clouds coming together. This paper introduces the first use of the SPH approximation formula's used to solve the Navier Stoke's equation for weakly compressible flow. Koshizuka et al. (1995) and Desbrun and Gascuel (1996) take Monaghan's paper and use it explicity for incompressible fluid flows using additional techniques developed by Monaghan (1989a). One of the first practical visual effects use of this technique was by Stora et al. (1999) in which he used the methods developed before, along with a method to calculate temperature diffusion across fluids, to animate lava flow. Up to this point, the research mainly focused on Newtonian fluids which changed with the introduction of the paper by Carlson et al. (2002) where he looks at simulating Non-Newtonian methods using newly designed methods for melting and flowing, along with adding an Elastic force to the calculations. Many papers followed on from this study with more advanced viscous and viscoelastic fluids research carried out by Mao (2006), Chang et al. (2009) and Clavet et al. (2005). Alongside these

papers Müller et al. (2003) and Müller et al. (2005) looked at improving on the current Newtonian Fluid techniques with the introduction of Dynamic Air Particles and multiple fluid forces such as Interface Tension Forces. These technqiues were optimised with multiple methods with a popular theme of using a method of Spatial Hashing developed by Teschner et al. (2003). The methods mentioned above are all based on solving versions of the Navier Stokes equations. There are however other particle based fluid techniques that depend on using functions such as the Leonard Jones Potential (Steele et al., 2004), that represent fluid behaviour on a purely function and non-physical level. Some current research has started to look at Viscoplastic fluids (fluids that change viscosity with force changes) by Paiva et al. (2009) adding a whole new dimension to Particle fluids.

3 SPH and Fluid Theory

When looking at fluid simulations there exists a whole host of algorithms that have been implemented, each with their own advantages and disadvantages. In this section the most prominent of these methods, using the Navier-Stokes Equations, is looked at and analysed. Solving these equations is then also looked at with the SPH approximation methods. First though is a review on the main two frameworks for representing fluid simulations.

3.1 Eulerian vs Langrangian

3.1.1 Lagrangian Method

The Lagrangian method looks at individual members of the fluid, sometimes called particles, and follows them as they move through space and time. The position of this particle can be plotted through time to give the pathline of the particle (Batchelor, 1973) (Lamb, 1994). In a simulation sense this is similar to a typical particle simulation, where each particle is assigned a number of attributes such as velocity, temperature etc...

Visualisation:
Sitting in a boat and drifting down a river.

3.1.2 Eulerian Method

The Eulerian method on the other hand looks at fluid motion that focuses on specific locations in space through which the fluid flows as time passes (Batchelor, 1973) (Lamb, 1994). In a simulation sense this method uses a grid system that divides the space into a number of rectangular cells, with each cell storing the relevant attributes inside such as velocity, pressure, density etc... Direct attributes can be read from each cell and neighbouring cells can be used for approximating the derivative (Win, 2007).

Visualisation:
Sitting on the bank of a river and watching the water pass a fixed location.

3.1.3 Advantages and Disadvantages of each method.

In Figure 1 is a table outlining the advantages and disadvantages of each method

	Eulerian	**Langrangian**
Advantages	Performance is independant of the number of particles. Fast.	Simulation size is not limited. Can be used for more than 1 type of fluid (water and air).
Disadvantages	Simulation detail limited to grid resolution. Simulation size limited to grid size.	Large number of particles needed for realistic simulation.

Figure 1: Advantages and Disadvantages of Eulerian and Lagrangian Methods.

Priscott (2010) looked at implementing a 2D Fluid Solver using a Eulerian method. In this paper the alternative approach is taken using purely Langrangian Methods and a range of SPH techniques.

3.2 Navier Stokes

The Navier Stokes equations provide a precise mathematical model for most fluids occurring in nature' (Stam, 2003). These equations look at solving unknowns for a fluid, in particular its new acceleration that allows for its new position to found. The equation in 3.2.1, is the Navier Stokes equation for Langrangian Fluid for weakly compressible flow. At first it looks particularly complex but when broken down into separate parts the basic principle behind its use is not as difficult as initially thought.

$$\rho\frac{d\mathbf{u}}{dt} = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f} \tag{3.2.1}$$

The left hand side of this equation is simply the density ρ of a particle multiplied by the acceleration $\frac{d\mathbf{u}}{dt}$. Knowing the acceleration means being able to integrate it to find the new position of the particle at the next timestep. Each term on the right hand side needs to be solved. The first of these is the Pressure term seen in equation (3.2.2).

$$-\nabla p \tag{3.2.2}$$

This term represents the gradient of the pressure field p and determines how the fluid moves with respect to this field. In this paper this term is referred to as the Pressure Force.

$$\mu\nabla^2\mathbf{u} \tag{3.2.3}$$

Equation (3.2.3) represents the Laplacian of the velocity field $\mathbf{u}$ multiplied by the viscosity μ, which is used to determine the force due to the viscosity of the fluid. In this paper this term is referred to as the Viscosity Force.

$$\mathbf{f} \tag{3.2.4}$$

The final term of this Navier Stokes formula is equation (3.2.4). This term represents all the external forces. As seen later, this will include forces such as Surface and Interface Tension forces and also forces such as gravity and wind.

Dealing with a Langrangian method, the most popular technique to solve these terms is using Smoothed Particle Hydrodynamics approximations, and is the method used in this project.

3.3 SPH Theory

Monaghan (1992) was one of the first people to solve the Navier Stokes equations using Smoothed Particle Hydrodynamics (SPH). His method focuses around solving each term of equation (3.2.1) using a collection of approximation formula's seen in Figure 3.3.

$$A(\mathbf{x}) = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{x} - \mathbf{x}_j, h) \tag{3.3.1}$$

$$\nabla A(\mathbf{x}) = \sum_j A_j \frac{m_j}{\rho_j} \nabla W(\mathbf{x} - \mathbf{x}_j, h) \tag{3.3.2}$$

$$\nabla^2 A(\mathbf{x}) = \sum_j A_j \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{x} - \mathbf{x}_j, h) \tag{3.3.3}$$

In these formula's $A(\mathbf{x})$ represents an attribute of a particle at location $\mathbf{x}$. This attribute can be anything that the user would like to solve, and in our case each term of the Navier Stokes equation plus terms such as the density ρ of each particle. W represents a smooth kernel weighting function that in simple terms adjusts each force depending on the distance between particles. The approximation formula's for each term, along with the Kernel functions, are looked at in much more detail in Section 5, however a quick look at how it works is looked at now.

For example, when solving the pressure term of Navier Stokes the Gradient of the pressure field needs to be found. In this case the term to solve is ∇p so the approximation equation (3.3.2) is used. All you have to do is substitute p for A and solve the formula by summing over all neighbouring particles, represented by j. This will give you the force due to pressure acting on the particle. Divide this by the density ρ of the particle and you have its acceleration from the pressure term. It is however important to note that this is a simplification of the process and adaptations have been made to solve some of these terms, so for full details please read Section 5.

3.4 Newtonian vs Non-Newtonian Fluids

When looking at simulating particle fluids a choice has to be made over what kind of fluids you want to be simulated. There are two main types of fluids, Newtonian and Non-Newtonian. As of yet there is no collection of equations that can successfully produce accurate results for Newtonian and Non-Newtonian fluids so a decision has to be made on which fluid this solver will simulate.

3.4.1 Newtonian Fluid

In a Newtonian fluid the relationship between the shear stress and the strain rate is linear (Mao, 2006). What this means is that the coefficient of viscosity is constant throughout the fluid and that they obey Newton's laws of motion, as implied by the name. Water is a good example of this.

3.4.2 Non-Newtonian Fluid

A Non-Newtonian Fluid on the other hand has a non-linear relationship between the shear stress and the strain rate (Mao, 2006). These fluids do not obey Newton's laws of motion and the viscosity of the fluid can change across the fluid depending on a multitude of factors such as temperature.

An example of some Non-Newtonian fluids are paint, cornflour (mixed with water) and some motor oil.

3.4.3 Chosen Fluid

This project looks at the implementation of Newtonian Fluids but keeps the idea of Non-Newtonian fluids in mind to allow for further work later, in the addition of a Non-Newtonian fluid solver to the same framework.

4 Design

When faced with designing a framework to allow for an efficient SPH simulation there a few key points that needed to be addressed and catered for.

- Must allow for different multiple fluids to be simulated in the same scene.
- Must be easily extendable to add and take away certain features in order to simulate a wide range of fluid effects.
- Must be able to allow simple interaction with Static Rigid Bodies and bounding areas.
- Must be efficient in passing large data structures around and force calculations must be optimised, in order for a fast simulation time.
- Data has to be easily exported to Houdini for Rendering and OBJ's easily imported for creating fluids.

In Section 4.1 there is a discussion on how these points have affected the overall design of the framework.

4.1 Class Diagram

Taking into account the points mentioned in the previous section, a Class Diagram was designed for the solver.

The diagram in Figure 2 shows the relationship between OpenGL and the solver. The GLWindow class is used to control all updates that happen each timestep. This information gets fed to the solver where the actual work gets done. The XML class is used to provide a communication link between an external package such as Houdini, by parsing XML files containing essential parameter data for the solver and the fluids being solved.

The diagram in Figure 3 on the other hand shows the SPHSolver's relationship with the other classes. As you can see, this class is the hub of all activity for the solver. For integration this class is linked to an Integrator class and for efficient neighbour searching this class is linked to a Spatial Hashing class. The SPHForce class is where all the force calculations take place, calculating the new acceleration for the solver. The implementation of all these classes is looked at in Section 5.

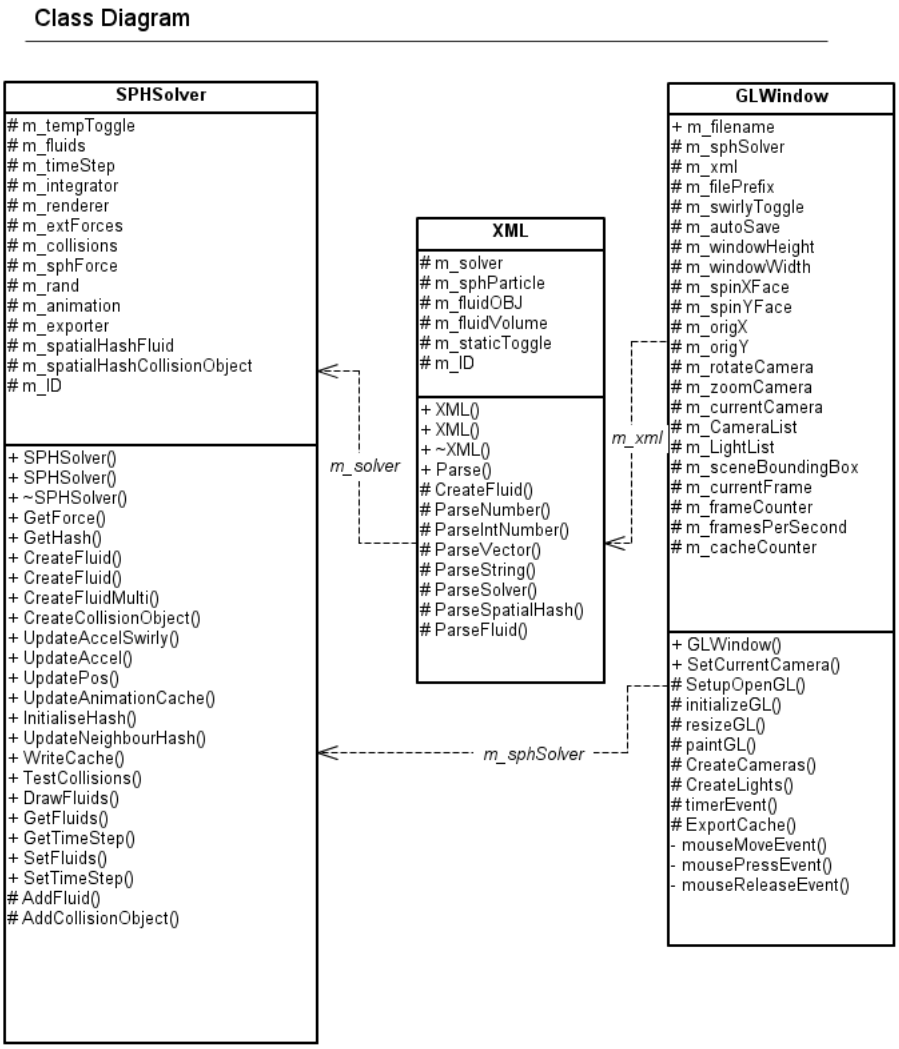


Figure 2: This class diagram show the interaction between GL Window and the solver components. The XML class is used to read XML config files which is then parsed by the solver. The GLWindow class controls updates to the Open GL window and is where the main functions such as UpdateAcceleration() are called.

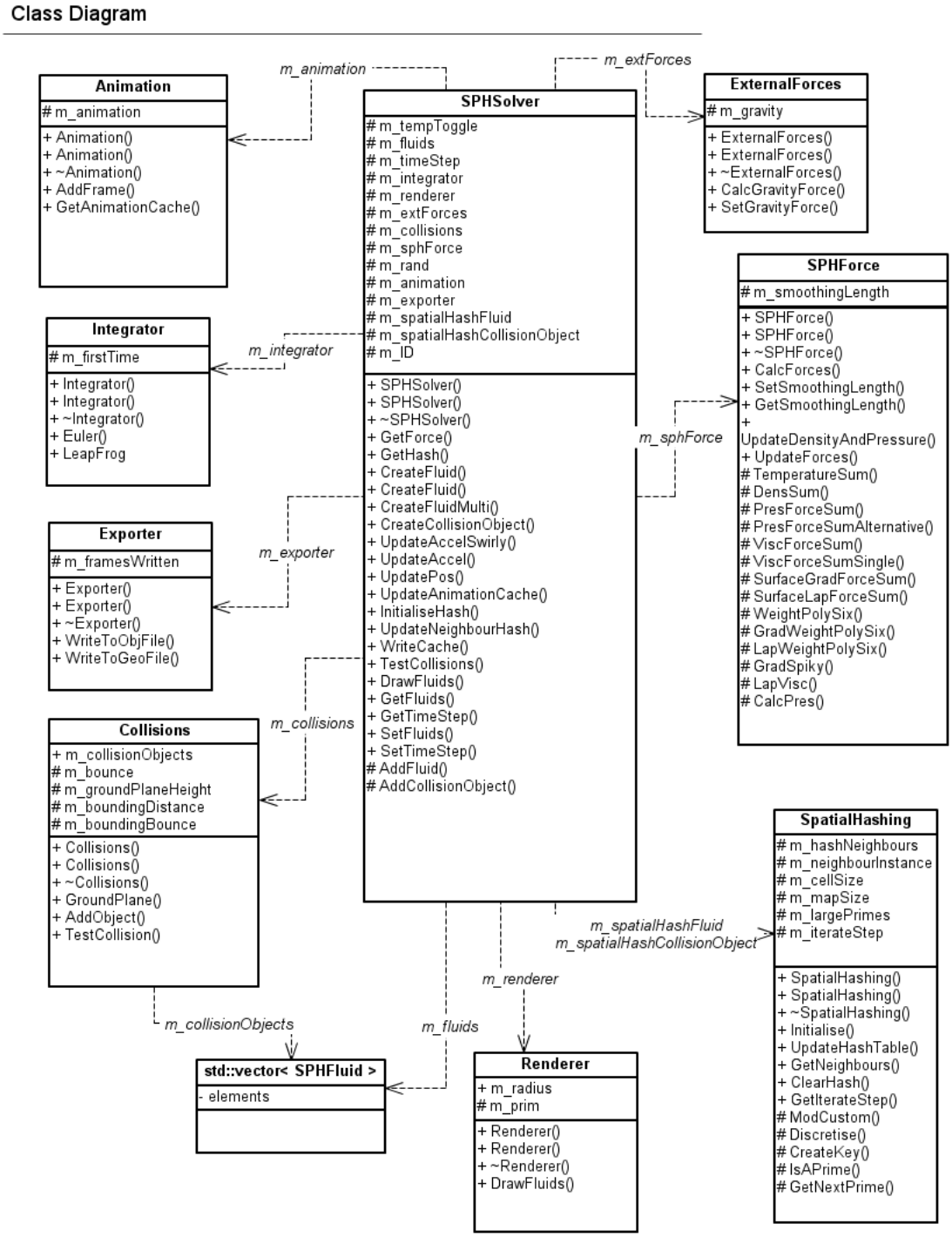


Figure 3: This class diagram shows the SPHSolver class's relationship with the other important classes. Object orientation is used to allow for changes to certain features without affecting other program fuctionality.

4.2 Representation of Fluid and Static RBD's

An important design decision that needed to made for this solver was how to represent the fluids and static bodies inside the program. Steele et al. (2004) uses a method that represents both the actual fluids and static bodies as an array of particles. What this method allows us to do is re-use methods such as the Spatial Hashing and Integration methods that will be implemented for the fluids on the static bodies. Finding neighbours becomes a simple task and allows for realistic interaction between both fluids and static bodies. This paper uses a similiar technique and is explained in more detail in Section 5.

4.3 Pipeline outline

When looking at how to set up the pipeline for this project a number of options are available.

- Stand alone application.
- Maya Plugin.
- Houdini Plugin/Digital Asset.

4.3.1 Stand alone application

To write a stand alone application that reads in OBJ's, simulates the scene and creates a rendered fluid surface is a huge task for the time period of this project. Just creating an efficient fluid surface using point splatting and marching cube takes a good deal of time. Therefore the option of a purely stand alone application is not viable.

4.3.2 Maya Plugin

When using Maya a lot of the work can be cut out, compared to a stand alone application, by writing a plugin. For example, no longer would a fluid surface algorithm need to be written as Maya already has a blobby surface that can create a fluid surface for you. Also, all the rendering functionality is readily available so the work load would allow sole focus on just the solver. However, Maya plugins are notouriously difficult to write and get working, so within the time frame available it is not a preferable option.

4.3.3 Houdini Plugin/Digital Asset

Using Houdini's HOM and Digital asset system however allows for all the benefits of Maya but with an API that is easy to utilise the power of the Digital asset system. Using Houdini is also a clever option as Houdini's current particle fluid system is limited in the range of effects it can achieve, so creating a new solver will help boost the functionality of Houdini. For example, currently Houdini is missing any temperature control, which is something that can be implemented with this new solver. Therefore for this project the solver is designed around a pipeline with Houdini.

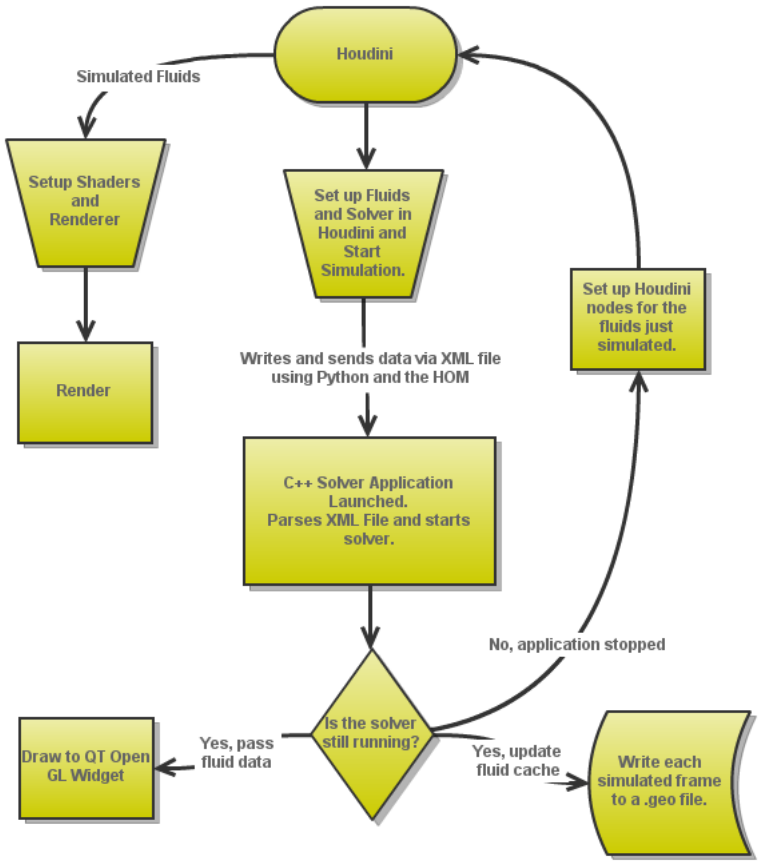


Figure 4: Flowchart representing the general outline of the pipeline.

In Figure 4 you can see an outline of the overall pipeline for the project. Houdini plays a crucial role in the pipeline as one of the ideas behind the project is to be able to improve the functionality of the existing particle fluids

inside Houdini. For this to be user and artist friendly the new solver needs to be able to be set up and run inside houdini, with the results fed back from the external program ready for the user to render. At no point should the user need to leave Houdini to run the solver or retrieve results. The way that this communication between the solver and Houdini is performed will be explained in detail in the section 6.

4.4 Pseudo-Code for program structure

The program structure for Particle Fluids plays a crucial part of how efficient a particle fluid simulation can be. When looking at solving the forces and retrieving neighbours the number of for loops and function calls needs to be kept at as a minimum, which in turn will allow for faster frame rates. In Figure 5 you can see a general outline of the whole program.

```
for eachFluid:
  CreateFluidParticles(); // Section 5.2
  CreateAndAddFluidToSolver();
end

for eachCollisionObject:
  CreateCollisionParticles(); // Section 5.2
  CreateAndAddCollisionFluidToSolver();
end

InitialiseSpatialHash(); // Section 5.5.1
InitialiseAnimationCache(); // Section 5.7

while(simulating):
  ClearAndRefillHash(); // Section 5.5.1
  UpdateAcceleration(); // Section 5.3
  UpdatePosition(); // Section 5.4
  TestCollisions(); // Section 5.6
  UpdateAnimationCache(); // Section 5.7
```

Figure 5: This pseudo code shows the high level structure of my program.

Most of the functions are self explanatory and relatively straight forward in terms of there inner workings (See Section 5 for a full breakdown of there workings). The first two loops deal with creating fluid objects (static or actual fluid). The next two function calls set up and initialise both the Spatial Hashing hash table and also the structure that allows for caching of the simulation. The next loop is where the bulk of the mathematics comes in, and is where the particle forces are calculated and moved accordingly. These functions are again explained in further detail in Section 5 but a quick look is given to the layout of one of the most important functions, UpdateAcceleration().

```
UpdateAcceleration():

  for eachFluid:
    for eachFluidParticle:
      GetNeighboursOfCurrentParticleFromHash(); // Section 5.5.1
      CalculateNewDensity(); // Section 5.3.1
      CalculateNewPressure(); // Section 5.3.2
    end
  end

  for eachFluid:
    for eachFluidParticle:
      GetNeighboursOfCurrentParticleFromHash(); // Section 5.5.1
      CalculateNewViscosityForce(); // Section 5.3.4
      CalculateNewPressureForce(); // Section 5.3.3
      CalculateNewSurfaceTensionForce(); // Section 5.3.5
      CalculateNewIntefaceTensionForce(); // Section 5.3.6
      CalculateForceFromGravity(); // Section 5.3.8
      UpdateTemperature(); // Section 5.3.7
    end
  end

  for eachFluid:
    for eachFluidParticle:
      CalculateSumOfForcesAndSetAcceleration(); // Section 5.3.10
    end
  end
```

Figure 6: The pseudo code for the Update Acceleration loop where the SPH Forces are calculated for each particle. Note that the reason the density and pressure is updated in a seperate loop is because the new density and pressure needs to be known for all the particles before the other forces can be calculated.

Figure 6 helps break down the inner workings of the function controlling all force calculations on the particles. It is important to note that in the actual program certain optimisations have been made in order to increase efficiency, but this Figure allows for a clearer view of the inner workings.

Note also how there are two for loops that iterate over the same objects. The reasons as you will see later are because of dependancies in the second loop for all the particles to have updated pressures and densities.

5 Implementation

In this section the implementation of the solver is looked at.

5.1 SPHParticles vs Particles

For a simple particle system each particle object only needs a small number of parameters. These parameters are acceleration, velocity, position and mass. With only these parameters and an integration method a very simple particle system can be created. However, when dealing with fluid particles each particle needs to contain more information than this. In this paper inheritance is derived and used from a simple Particle class to create a class called SPHParticle. This class inherits the basics needed for a particle system but also includes the attributes seen in Figure 7 . In this paper, the decision was made to store all these attributes with each particle rather than with each fluid. This increases memory but allows for a wider variety of fluid effects to be implemented. For example, when dealing with temperature each particle of the fluid should have an individual temperature attribute so that diffusion of heat can occur across an individual fluid and multiple fluids.

5.2 Creating Fluids

For a sophisticated solver, a method that allows for a range of objects to turn into fluid needs to be implemented. By using Macey (2010)'s Graphic Library, OBJ objects are able to be imported into the scene. The vertices that make up this OBJ are then used as positions for the particles that make up the fluid. Each of these particles are set up to hold the variables in Figure 7 as specified by the user. It is important to note that the OBJ's that the user imports should have been filled with points (by using the Points from Volume SOP in Houdini for example (Inc., 2010)) to allow for a more realistic fluids to be created.

When these SPHParticles have been created an SPHFluid object is created using the array of these particles. Multiple SPHFluid Objects can be created using this method and an array of these fluid is stored in the SPH-Solver Object.

Setting up these parameters is discussed in the section on pipeline, Section 6.

SPHParticle attribute	Symbol	Stored as	Units
Position	$\mathbf{x}$	ngl::Vector	m
Velocity	$\mathbf{u}$	ngl::Vector	m/s
Acceleration	$\mathbf{a}$	ngl::Vector	m/s^2
Accumalated Forces	$\mathbf{f}$	ngl::Vector	N/m^3
Mass	m	ngl::Real	Kg
Pressure	p	ngl::Real	Pa
Viscosity	μ	ngl::Real	Ns/m^2
Temperature	T	ngl::Real	^oC
Density	ρ	ngl::Real	Kg/m^3
Rest Density	ρ_0	ngl::Real	Kg/m^3
Gas Constant	k	ngl::Real	Nm/Kg
Colour Surface Tension	c^s	ngl::Real	1
Colour Interface Tension	c^i	ngl::Real	1
Diffuse Constant	c	ngl::Real	1
Ideal Gas Constant	α	ngl::Real	1
Collision Radius	r	ngl::Real	m

Figure 7: The attributes stored with each SPHParticle. Note that the Collision Radius only applies to static objects and not fluids.

5.2.1 Calculating the mass

One of the most important parameters for the fluid is the mass of each of the particles. In the final system however you are unable to set this parameter, the reasons why will be explained now. You will notice that a parameter that is not stored on a particle but a parameter you are able to set for each fluid is the Volume. Kelager (2006) uses this Volume, the density of a particle and the number of particles making up a fluid to calculate the mass of an individual particle by using equation (5.2.1).

$$m = \rho \frac{V}{n} \tag{5.2.1}$$

.

Using this method is much more practical than specifying the mass as it allows the user to get useful results without having to adjust the mass and calculate this equation themselves. When a fluid is being created this parameter is easily set as the NGL OBJ class is able to calculate the total number of vertices (particles) and the density and volume will have been set by the user already. It is important to note that this equation is only

calculated once on fluid creation and hence the mass of each particle will remain constant for the rest of the simulation.

5.3 Calculating and Applying Forces

With an array of fluids the next step is to start to solve the terms of (3.2.1) for each iteration of the solver, and eventually find the acceleration of each particle. To do this the SPH approximation equations (3.3.1), (3.3.2) and (3.3.3) are utilised. For example, for an attribute such as the density the general attribute $A(\mathbf{x_i})$ is substituted for the density attribute ρ_i. The class SPHForce is used to implement these equations and perform these calculations.

When it comes to how the smoothing kernels work and are calculated this is left to be discussed in detail in Section 5.3.9.

5.3.1 Calculating the Density

In the SPH approximation equations you can see that the density of the particles is used in each. Therefore it is important that all the particle densities are updated before solving the other terms of the Navier Stokes equation. To find the Density there are two popular approaches that can be used. The first is called the 'density summation' technique and can be seen in equation (5.3.2) (Monaghan, 1992). This method works by directly applying the SPH approximations to the density ρ itself. This method technically ignores the Navier Stokes method for finding the density, which is where the second technique comes in. This other technique is called the 'continuity density' and as the name suggests its uses the continuity equation when making SPH approximations, see equation (5.3.1). This uses the derivative SPH approximation to solve and then needs to be integrated in order to find the actual density of each particle. There are benefits and disadvantages of each approach but for the simulation that is presented in this paper the 'density summation' works just fine (Kelager, 2006).

$$\frac{\delta\rho}{\delta t} = -\rho\nabla\mathbf{u} \tag{5.3.1}$$

$$\rho(\mathbf{x}_i) = \sum_j m_j W_{default}(\mathbf{x}_i - \mathbf{x}_j, h) \tag{5.3.2}$$

In this 'density summation' equation, and also in the other force equations, j represents all the particles within the smoothing length h of particle

$\mathbf{x}_i$). Finding the particles that make up j is talked about later in the section on Optimisation, Section 5.5.1.

As mentioned briefly in 5.3 this equation has been derived from using the SPH equation (3.3.1) and is only affected by m_j as the two ρ_j cancel each other out.

5.3.2 Calculating the Pressure Per Particle

When calculating the Pressure term of equation (3.2.1) the pressure p_i of every particle needs to be known before solving the force itself. Therefore, when calculating the new density for each particle its new pressure is also updated. Desbrun and Gascuel (1996) introduces a modified version of the ideal gas state equation in order to find the pressure, equation (5.3.3).

$$p_i = k\left(\rho_i - \rho_0\right) \tag{5.3.3}$$

In this equation k is a gas constant and ρ_0 is the rest density of the fluid. This equation can return negative and positive values, which allows for an attractive or repulsive force that is minimised as the density gets nearer the rest density. This is the exact nature needed so is perfect for the job.

5.3.3 Calculating the Pressure Force

Knowing the new density and new pressure for each particle allows us to start solving the specific terms of the Navier Stokes equation (3.2.1). This sections looks at solving the pressure term to find its corresponding force. Again this can be done by utilising the SPH attribute equations and in this specific case equation (3.3.2). This is because the gradient of the pressure field needs to be found and not just the pressure field. Following the same method as finding the density (a simple substitution into the SPH approximation equation) gives us equation (5.3.4).

$$\mathbf{f}_i^{presure} = -\sum_{j \neq i} p_j \frac{m_j}{\rho_j} \nabla W\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.3.4}$$

However, using this equation leads to problems. The main problem encountered is that the pressure force is not symmetrical. Kelager (2006) expresses this problem nicely and uses the example of looking at only two particles. Using this equation 'the first particle only uses the pressure at the second particle to compute its pressure force' and vice versa for the second particle. 'Because the pressures at the particles are not equal in general the pressure force will be assymmetric, and the action-reaction law will not be conserved' (Kelager, 2006).

To fix this problem a method is used from one of the original papers by Monaghan (1992). In this paper Monaghan introduces the concept that it is better to rewrite formulae with the density placed inside the operators. This is sometimes called the second golden rule of SPH and after more mathematical manipulation a higher accuracy can be obtained on the gradient of our pressure field using equation (5.3.5).

$$\mathbf{f}_i^{presure} = -\rho_i \sum_{j \neq i} \left(\frac{p_i}{p_i^2} + \frac{p_j}{p_j^2} \right) m_j \nabla W_{pressure} \left(\mathbf{x}_i - \mathbf{x}_j, h \right) \tag{5.3.5}$$

5.3.4 Calculating the Viscosity Force

In the same loop that calculates the pressure force the solver is also able to calculate and update the viscosity force (equation (3.2.3) for each particle, and find the force it exerts into the simulation. If the simple SPH approximation substitution method is used to calculate this term a similiar problem to the pressure term is encountered, where the forces are assymetric. In this case Müller et al. (2003) proposes a method that allows for the velocity fields to be symmetrised, possible due to the fact that the viscosity forces don't depend on absolute velocities but velocity differences. The equation that is used in this paper is equation (5.3.6).

$$\mathbf{f}_i^{viscosity} = \mu \sum_{j \neq i} \left(\mathbf{u}_j - \mathbf{u}_i \right) \frac{m_j}{\rho_j} \nabla^2 W_{viscosity} \left(\mathbf{x}_i - \mathbf{x}_j, h \right) \tag{5.3.6}$$

In this equation μ is the viscosity coefficient of the fluid. The higher this value the more viscous the fluid, and vice versa. The viscosity of a fluid can be thought of as a fluids resistance to flow. This resistance comes from internal friction between fluid molecules when a fluid flows, leading to a decrease in its kinetic energy due to heat loss.

5.3.5 Calculating the Surface Tension Force

Now that the two internal forces have been solved the next step is to look at solving the external forces. One of these forces, although not strictly part of the Navier Stokes equation, is the Surface Tension Force (Müller et al., 2003). This force represents how fluid particles on the surface are attracted towards the bulk of the particles in the fluid. For viscous fluids such as honey this force is essential for accurate simulations as honey strongly sticks together as one and rarely splits into multiple pools of fluid. The desired result is for the 'surface tension force to act in the direction of the inward surface normals

towards the fluid, where they bind the fluid surface together' (Kelager, 2006). This force 'will flatten the surface curvature by minimising the surface area'. This can be seen best in Figure 8.

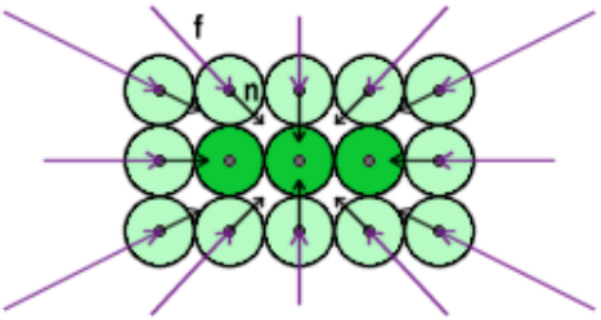


Figure 8: This shows the surface tension force f as calculated by looking at the inward surface normals n of the fluid particles. Figure from (Kelager, 2006)

(Müller et al., 2003) proposes the formula in (5.3.7) to calculate this Surface force. Equation (5.3.8) is used to calculate n.

$$\mathbf{f}_i^{surface} = -\sigma \frac{\mathbf{n}_i}{\|\mathbf{n}_i\|} \sum_j \frac{m_j}{\rho_j} \nabla^2 W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.3.7}$$

$$\mathbf{n}\left(\mathbf{x}_i\right) = \sum_j \frac{m_j}{\rho_j} \nabla W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.3.8}$$

In these equations σ is used to represent the tension force, which should be defined depending on the two surfaces being tested between, ie. honey in water will have a different surface tension to honey in air. Also, what is important to watch out for here is the fact this equation involves dividing by the length of a vector $\|n_i\|$. As particles get closer together this tends towards zero and will start introducing instability into the system, causing inaccurate simulations. To control this a surface tension threshold β is defined that says if $\|n_i\| \leq \beta$ then set the $\mathbf{f}_i^{surface} = 0$.

There are similiarites here to the pressure and viscosity force equations and you can see that the $default$ smoothing kernel is used when calculating the 'influence' from neighbouring particles. This formula is implemented in a similar light to the pressure and viscosity formula's.

5.3.6 Calculating the Interface Tension Force

With the introduction of multiple fluids to the simulation it is important to also bear in mind the force between two fluids. In the real world, fluids either mix or don't mix depending on their polarity. Müller et al. (2005) uses this fact to design an Interface tension force that describes how multiple fluids behave when they coerce with each other. This force calculation is almost identical to the surface tension force except for the introduction of one new variable c^i, as you can see in equation (5.3.9).

$$\mathbf{f}_i^{interface} = -\sigma \frac{\mathbf{n}_i}{\|\mathbf{n}_i\|} \sum_j \frac{m_j}{\rho_j} \nabla^2 c^i W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.3.9}$$

This new variable controls whether the fluid is polar or non-polar and hence determines whether two fluids mix or not. Two polar fluids will mix together so taking into account the negative sign at the start of equation (5.3.9) all polar fluids c^i are set with the value of -0.5. This means that positive force's will be returned allowing the two fluids to mix. On the contrary, a polar and non-polar fluid will not mix so a non-polar fluids colour field c^i is set to 0.5 in order for a repulsive force between these two fluids. Note that the normal n_i is calculated in the same way using (5.3.8).

5.3.7 Adding Temperature

A very basic but effective temperature solver is also implemented in this project. Müller et al. (2005) presents a method that uses the SPH approximations and the Ideal Gas Law. This method calculates the temperature change rate using the particles temperature value and the Gradient SPH approximation, as seen in equation (5.3.10). The reasons for being able to do this is because of equation (5.3.11), stating how an attribute evolves due to diffusion, with the attribute in our case being temperature. This equation is solved as before by using the Gradient SPH approximation in equation (5.3.10).

$$\frac{dT}{dt} = c \sum_j m_j \frac{T_j - T_i}{\rho_j} \nabla^2 W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.3.10}$$

$$\frac{dA}{dt} = c \nabla^2 A \tag{5.3.11}$$

The constant c in equation (5.3.10) represents a Diffuse constant that allows for fluids that transfer heat differently. A in equation (5.3.11) simply represents any attribute.

Having the rate of change for the temperature, the new temperature is calculated for the particle with a single integration using a simple Euler Integration step. Müller et al. (2005) then uses the Ideal Gas Law with a Ideal Gas Constant α to produce a new rest density for the particle as in equation (5.3.12).

$$\rho_0 \sim \frac{1}{V} \sim \frac{\alpha}{T} \tag{5.3.12}$$

This means that as the temperature increases the rest density decreases, just as in nature. For example, heating water decreases the rest density making it turn from liquid to gas.

5.3.8 Calculating the Gravity Force

The last force that needing to be calculated is the force of gravity. To calculate this force is trivial using Newtons Second Law of Motion giving formula (5.3.13).

$$\mathbf{f}_i^{gravity} = \rho_i \mathbf{g} \tag{5.3.13}$$

5.3.9 Smoothing Kernels

For all the force equations just talked about a variety of smoothing kernel functions with the form $W(\mathbf{x}_i - \mathbf{x}_j, h)$ are used. This section looks at the reason and importance for these functions.

These smoothing kernel functions are the basis behind why fluid can be simulated using particles and the approximation methods. The principle behind them is a function that determines how much 'influence' a particle has on the other particle being tested against. These functions use a distance refered to as h and is called the smoothing length. If the distance between two particles falls within this smoothing length then a non-zero value is returned from the smoothing function. There are specific rules that must be followed when designing these functions in order to have a useable function for SPH. These rules where first introduced by Monaghan (1992) and are represented in equations (5.3.14), (5.3.15) and (5.3.16). In the following smoothing kernel functions r is used to represent the distance between particles $\mathbf{x}_i - \mathbf{x}_j$.

$$\int_\Omega W(\mathbf{r}, h)\, d\mathbf{r} = 1 \tag{5.3.14}$$

$$W(\mathbf{r}, h) \geq 0 \tag{5.3.15}$$

$$W(\mathbf{r}, h)\, d\mathbf{r} = W(-\mathbf{r}, h) \tag{5.3.16}$$

Equation (5.3.14) ensures that the unit integral maxima and minima are not enhanced, and that the kernel must be normalised. Equation (5.3.15) ensures that only positive results are returned and that the smoothing kernel is averaging function (Koshizuka et al., 1995). The last equation that needs to be met is equation (5.3.16). This equation ensures that the function is even, which is important as it ensures that 'rotational symmetry is enforced. This is useful to ensure invariance under rotations of the coordinate system.

The first SPH golden rule (Monaghan, 1992) says that if a new interpration of an SPH equation is to be found then it is best to assume the kernel is a Gaussian. However, as seen later, the Gaussian does not always give the best results and more accurate proposals have been made.

Bearing these 3 essential rules in mind, a collection of accurate smoothing kernel functions have been proposed which are looked at now.

The first smoothing function that is looked at is $W_{default}(\mathbf{x}_i - \mathbf{x}_j, h)$. The equations for this (including the gradient and laplacian) are equations (5.3.17), (5.3.18) and (5.3.19). This kernel function was first used and proposed by Müller et al. (2003) and is also know as the 6th degree polynomial kernel.

$$W_{default}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} \left(h^2 - \|\mathbf{r}\|^2\right)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \|\mathbf{r}\| > h \end{cases} \tag{5.3.17}$$

$$\nabla W_{default}(\mathbf{r}, h) = -\frac{945}{32\pi h^9}\mathbf{r}\left(h^2 - \|\mathbf{r}\|^2\right)^2 \tag{5.3.18}$$

$$\nabla^2 W_{default}(\mathbf{r}, h) = -\frac{945}{32\pi h^9}\mathbf{r}\left(h^2 - \|\mathbf{r}\|^2\right)\left(3h^2 - 7\|\mathbf{r}\|^2\right) \tag{5.3.19}$$

A noticeable quality of the default kernel is its similiarity to the Gaussian, which helps stick to the first Golden Rule of SPH. This can be seen clearly in Figure 9 and makes it the ideal kernel function to use when calculating the density, and also the surface and interface tension forces.

For the density for example, this simple default kernel allows for particles that are closer to the particle being tested to have more influence the closer they are, just as you would expect.

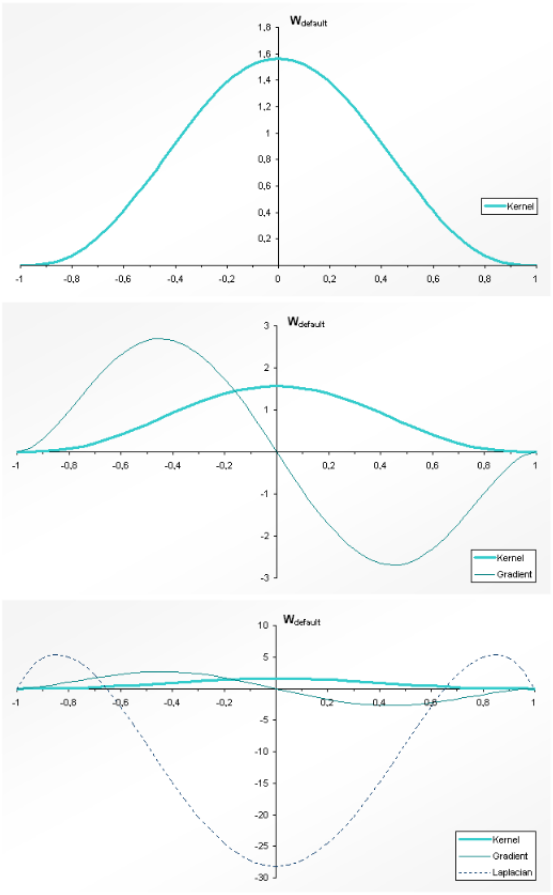


Figure 9: The Default Kernel in one dimension, with its Gradient and Laplacian. h is set to 1. Notice how each function tends to zero as it nears h (1 in this case). This makes sense as it says that if a particle is further than the smoothing length then the function returns 0 and has no 'influence' on the particle being tested. Figure from (Kelager, 2006)

However, when dealing with calculating the pressure and viscosity term problems arise when using this kernel. Desbrun and Gascuel (1996) talks about the specific problem for the pressure term. When calculating the pressure term the Gradient of the Smoothing Kernel needs to be used seen in equation (5.3.18). The problem occurs when the distance between particles tends towards zero. As you can see in figure 9 the gradient function tends towards zero as the distance between particles tends towards zero, $\lim_{\|\mathbf{r}\|\to 0} = 0$. This means that in high pressure regions (where there is a high density of particles) instead of being repulsed they cluster. Müller et al. (2003) solves

this problem with the introduction a new specific smoothing kernel for the pressure which can be seen in equations (5.3.20), (5.3.21), (5.3.22) and also Figure 10.

$$W_{pressure}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\mathbf{r}\|)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \|\mathbf{r}\| > h \end{cases} \tag{5.3.20}$$

$$\nabla W_{pressure}(\mathbf{r}, h) = -\frac{45}{\pi h^6} \frac{\mathbf{r}}{\|\mathbf{r}\|} (h - \|\mathbf{r}\|)^2 \tag{5.3.21}$$

$$\nabla^2 W_{pressure}(\mathbf{r}, h) = -\frac{90}{\pi h^6} \frac{1}{\|\mathbf{r}\|} (h - \|\mathbf{r}\|)(h - 2\|\mathbf{r}\|) \tag{5.3.22}$$

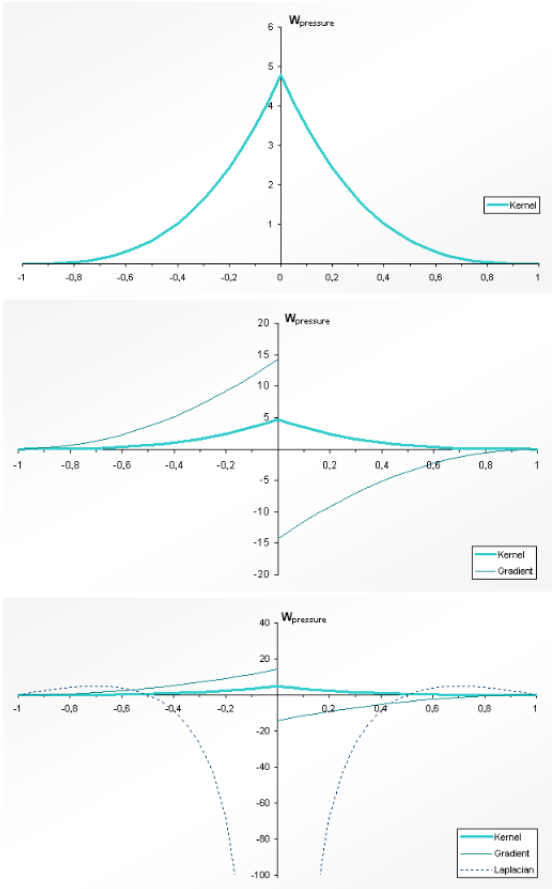


Figure 10: The Kernel used for the Pressure term. It is important to note that now the Gradient of the kernel function no longer tends towards zero, which helps avoid clustering. Figure from (Kelager, 2006)

Now when two particles are very close together the gradient of the kernel function shows that $\lim_{\|\mathbf{r}\|\to 0} = \frac{45}{\pi h^6}$. This ensures that clustering is avoided by a repulsion force at zero, allowing for a better representation of how pressure affects the fluid particles.

The final smoothing kernel function that is dealt with is the one used for the viscosity term. Again, the default kernel is not suitable for our requirements so a new kernel needs to be designed. In this case Müller et al. (2003) proposes another kernel function, as seen in equations (5.3.23), (5.3.24), (5.3.25).

$$W_{vicosity}(\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{\|\mathbf{r}\|^3}{2h^3} + \frac{\|\mathbf{r}\|^2}{h^2} + \frac{h}{2\|\mathbf{r}\|} - 1 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \|\mathbf{r}\| > h \end{cases} \tag{5.3.23}$$

$$\nabla W_{vicosity}(\mathbf{r}, h) = \frac{15}{2\pi h^3}\mathbf{r}\left(-\frac{3\|\mathbf{r}\|}{2h^3} + \frac{2}{h^2} - \frac{h}{2\|\mathbf{r}\|^3}\right) \tag{5.3.24}$$

$$\nabla^2 W_{vicosity}(\mathbf{r}, h) = \frac{45}{\pi h^6}\mathbf{r}(h - \|\mathbf{r}\|) \tag{5.3.25}$$

The important function here is the Laplacian of the function as this is what is used when finding the viscosity term. As you can see in equation (5.3.25) and Figure 11 the function will always return a positive value. This is important as now the viscosity only acts as a damping force. If negative values are returned then energy is introduced into the fluid, which will cause inaccuracy and instability. For this reason both the default kernels and pressure kernels are not useable due to the fact that there Laplacian functions return negative values as well as positive values.

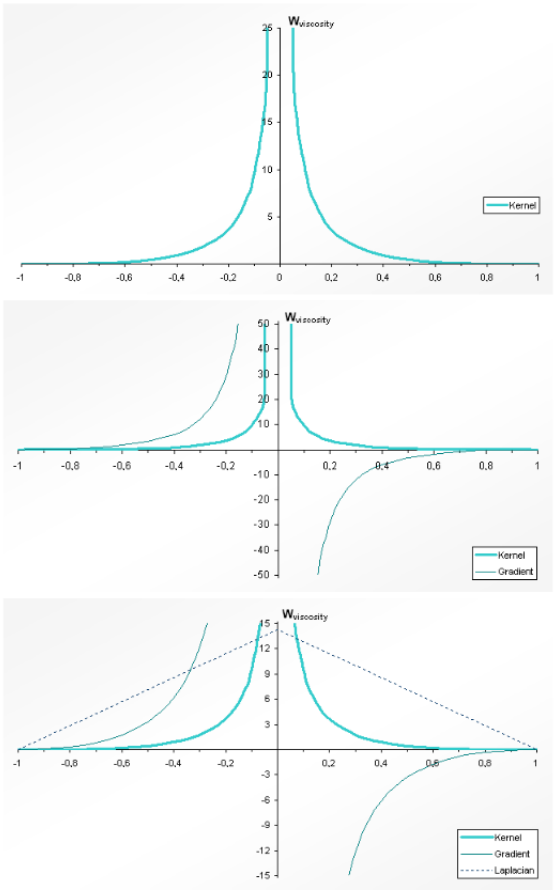


Figure 11: The Viscosity Kernel function. The important feature of this is that the Laplacian function is always positive meaning that the viscosity force is only ever a damping force and does not introduce energy into the fluid. Neither the default or pressure kernel have this feature, and hence why this kernel is used. Figure from (Kelager, 2006)

5.3.10 Calculating Acceleration

After running through both loops in Figure 6 for all particles a selection of forces will have been found acting on each particle. From this the new acceleration is able to be calculated for each particle by using the formula (5.3.26).

$$\mathbf{a} = \frac{d\mathbf{u}}{dt} = \frac{\mathbf{F}}{\rho} \tag{5.3.26}$$

This formula follows Newton's second law of motion and is a re-arrangement of the Navier Stokes equation found in equation (3.2.1). $\mathbf{F}$ represents the sum of all the forces, ie, the sum of the viscosity, pressure, surface tension and interface tension forces. The acceleration is updated like this for every particle in our scene.

5.4 Integration Methods

Knowing the new acceleration for each particle allows us to look at integrating this to find the new velocity and then the new position for each particle. In this paper two method's have been implemented, the traditional explicit Euler Method and the implicit Leap Frog Method.

5.4.1 Euler Method

The explicit Euler method is one of the most simple integration methods but unfortunately it is also the most unstable. Equations (5.4.1) and (5.4.2) show the method.

$$\mathbf{u} = \mathbf{u} + \mathbf{a}dt \tag{5.4.1}$$

$$\mathbf{x} = \mathbf{x} + \mathbf{u}dt \tag{5.4.2}$$

In these equations $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{a}$ are the position, velocity and acceleration respectively. The problem with this method is that very small timesteps dt need to be used for the simulation to be accurate and stable.

5.4.2 Leap Frog Method

An alternative method to using Euler, and a popular method for SPH simulations (Paiva et al., 2009), is to use the Leap Frog Scheme. The reason for this name relates to the nature of how the method works, with the velocities leaping to future timesteps to find current positions. The steps to get from

the acceleration to the new position can be seen in equations (5.4.3) and (5.4.4).

$$\mathbf{u}_{t+\frac{1}{2}\Delta t} = \mathbf{u}_{t-\frac{1}{2}\Delta t} + \Delta t \mathbf{a}_t \tag{5.4.3}$$

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \Delta t \mathbf{u}_{t+\frac{1}{2}\Delta t} \tag{5.4.4}$$

This method works by using future half-step velocities to find a more accurate and stable value for the position.

As this is an implicit method a specific formula is needed to find the first velocity at a half timestep. This is found using equation (5.4.5).

$$\mathbf{u}_{-\frac{1}{2}\Delta t} = \mathbf{u}_0 - \frac{1}{2}\Delta t \mathbf{a}_0 \tag{5.4.5}$$

As this method only calculates the velocities at half-steps the actual velocity at the timesteps is not known. If it is decided that this velocity needs to be known at whole time-steps then the equation (5.4.6) can be used.

$$\mathbf{u}_t \approx \frac{\mathbf{u}_{t-\frac{1}{2}\Delta t} + \mathbf{u}_{t+\frac{1}{2}\Delta t}}{2} \tag{5.4.6}$$

This method proves to be much more accurate and stable at larger timesteps than Euler. Also with its relative simplicity it is chosen to use this over Euler in the majority of cases.

5.5 Optimisations

When dealing with a large number of fluid particles, especially when introducing multiple fluids, the simulation can be very inefficient if optimisations are not made. The most important optimisation is how to find the array of neighbours j of a particle. Without a method to find neighbours all particles need to be checked against every other particle, which is of course very efficient. A naive estimate at the time complexity of how fast an SPH Simulation is using this method is $O(n^2)$) (Kelager, 2006). Müller et al. (2003) suggests a grid based method, which even though increases efficiency it constrains the simulation to a specific area. One of the benefits of using a Langrangian method for Fluid Simulation is this lack of a bounding volume so using this grid based method is not an advantage to us. Instead a method is used known as spatial hashing introduced by Teschner et al. (2003), which decreases the time complexity to $O(nm)$) where m is the average number of neighbours found. The more uniform the distribution of particles the lower m will be, and the faster the SPH simulation will run.

5.5.1 Spatial Hashing

Spatial Hashing falls under the category of fast NNS (Nearest Neighbour Search) algorithms and in theory is bound by $O(1)$ as discussed earlier. The basic idea behind how Spatial Hashing works is similiar to that of a normal hash table. Positions in a scene are hashed to create a key, which specifies its position in the hash table. The idea is that positions close to each other are hashed to the same bucket in the hash table. Therefore the theory suggests that to find the neighbours of a particle at a certain position they can be looked up at its hashed position in the table and in the same bucket should be its neighbours. The implementation of this is now looked at.

The first thing that is looked at is how to take a particles location and create a hash key from it. The complexity behind this algorithm is highly dependant on how well this hash function generates unique keys, and how fast keys are generated (Kelager, 2006). The importance of having unique keys is to avoid having particles at significantly different locations hashing to the same cell in the hash table.

Teschner et al. (2003) recognises this problem and has proposed the hash function that can be seen in equation (5.5.1)

$$hash\left(\hat{\mathbf{x}}\right) = \left(\hat{\mathbf{x}}_x p_1 \ xor \ \hat{\mathbf{x}}_y p_2 \ xor \ \hat{\mathbf{x}}_z p_3\right) \ mod \ n_H \tag{5.5.1}$$

At first this funtion looks quite complex but when each part is broken down it is easier to understand. The first thing to note is the definiton of $\hat{x}$. This is a discretising function that takes a vector with floating point values and creates a vector with integer values based on cell size l. See equation (5.5.2).

$$\hat{\mathbf{x}}\left(\mathbf{x}\right) = \left(\lfloor \mathbf{x}_x / l \rfloor, \lfloor \mathbf{x}_y / l \rfloor, \lfloor \mathbf{x}_z / l \rfloor\right)^T \tag{5.5.2}$$

It is useful to note that here that our cell size l can be set to the smoothing length h as particles that lie outside of the smoothing length don't affect the force calculation.

$$l = h \tag{5.5.3}$$

Another unknown in equation (5.5.1) is p_1, p_2, p_3. These are just large prime numbers and are the same values used by Kelager (2006) and can be found in equations (5.5.4), (5.5.5) and (5.5.6).

$$p_1 = 73,856,093 \tag{5.5.4}$$

$$p_1 = 19,349,663 \tag{5.5.5}$$

$$p_1 = 83,492,791 \tag{5.5.6}$$

The final unknown in equation (5.5.1) is n_H. This is the size of the table and is calculated this using equation (5.5.7) (Teschner et al., 2003).

$$n_H = prime(2n) \tag{5.5.7}$$

In this equation n is the total number of particles in our scene and $prime(x)$ is a simple function that returns the next prime after x.

Knowing how to take a particle position and run it through our hash function, the next step is to fill a hash style table for all the hashed particles. In this project the Standard Template Library's multimap is used. This is perfect for the job as it allows for multiple objects to be hashed to the same cell. This is important as all the neighbours need to be found and not just one neighbour.

Before neighbours of a specific particle can be searched for the hash table first needs to be filled with all the particles. For every particle the hash table is filled using equation (5.5.8)

$$hash\_table[hash\left(\hat{\mathbf{x}}\left(\mathbf{x}_i\right)\right)] = Particle_i \tag{5.5.8}$$

Having filled the table with all our particles the neighbour particles at any location can be found. A simple way to do this would be take the location of the particle of which you want to find neighbours and simply run it through the hash function and return all the particles that hash to the same cell. However problems occur here as not all neighbours of a particle will get hashed to the same cell. Therefore this algorithm iterates over positions of a bounding area, returning neighbours of all the hashed iterated positions, and finally double checking which of these neighbours is actually a neighbour of our query particle. Equation (5.5.9) gives us the two corner points of our bounding area (Kelager, 2006).

$$BB_{min} = \hat{\mathbf{x}}\left(\mathbf{x}_Q - (h,h,h)^2\right) \ , \ BB_{max} = \hat{\mathbf{x}}\left(\mathbf{x}_Q + (h,h,h)^2\right) \tag{5.5.9}$$

The symbol $\mathbf{x}_Q$ represents our query particle, ie. the particle that neighbours need to found for. BB_{min} and BB_{max} are the two corner points of the bounding area. The next step is to iterate over this bounding area. This can be hard to get your head around so an example of this iteration function can be seen in the Appendix with Figure 23

Each iteration a position pos_D is used which is used to build up a dynamic list L of all the possible neighbour particles, (5.5.10).

$$L = hash\_table[hash\,(pos_D)] \tag{5.5.10}$$

As mentioned earlier, the particles that fill L are not all neccessarily neighbours so a check is made using equation (5.5.11) to remove neighbours that don't lie in the smoothing length of the query particle.

$$\|\mathbf{x}_Q - \mathbf{x}_j\| \ \leq h \tag{5.5.11}$$

5.5.2 Particle Velocity Correction

One problem that is occasionally faced when implementing SPH simulations is particle interpenetration problems. To solve this Monaghan (1989a) proposed a technique called XSPH. This method corrects the velocity for each particle by computing an average velocity of the nieghbouring particles (Paiva et al., 2009).

$$\mathbf{v}_i = \mathbf{v}_i + \epsilon \sum_j \frac{2m_j}{\rho_i + \rho_j} \left(\mathbf{v}_j - \mathbf{v}_i\right) W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.5.12}$$

The variable ϵ is just a global parameter that determines how much correction you want to apply. Pushing this value too high will take away from the physical correctness of the simulation. The effect achieved is that of giving the particles a more orderly flow especially when dealing with a weakly compressible flow (Paiva et al., 2009). This algorithm is performed between integrating the acceleration and integrating the velocity.

5.5.3 Artificial Viscosity

Another problem that can be faced is when the accelerations being produced are very high. An example of when this can happen could be when a fluid is compressed by a moving static object, increasing the pressure considerably, which in turn will mean a high acceleration away from the area of high pressure. To help tackle this problem an artificial viscosity can be introduced. Monaghan (1992) first presented this technique and uses equations (5.5.13), (5.5.14) and (5.5.15).

$$\frac{d\mathbf{u}_i}{dt} = \frac{d\mathbf{u}_i}{dt} - \sum_j m_j \prod_{ij} \nabla_i W_{default}\left(\mathbf{x}_i - \mathbf{x}_j, h\right) \tag{5.5.13}$$

$$\prod_{ij} = \begin{cases} -\frac{2\alpha\mu_{ij}c}{\rho_i+\rho_j}, & (\mathbf{u}_i - \mathbf{u}_j)\cdot(\mathbf{x}_i - \mathbf{x}_j) < 0 \\ 0, & (\mathbf{u}_i - \mathbf{u}_j)\cdot(\mathbf{x}_i - \mathbf{x}_j) \geq 0 \end{cases} \tag{5.5.14}$$

$$\mu_{ij} = \frac{h\,(\mathbf{u}_i - \mathbf{u}_j)\cdot(\mathbf{x}_i - \mathbf{x}_j)}{|\,(\mathbf{x}_i - \mathbf{x}_j)\,|^2 + 0.001h^2} \tag{5.5.15}$$

In this paper the same method is implemented as Monaghan (1992). In equation (5.5.14) α represents a bulk viscosity that is defined by the user and c represents the speed of sound. The speed of sound is used as this represents the fastest velocity of a wave propagation in that medium (Paiva et al., 2009).

This formula is performed after the acceleration has been calculated, using the formula in Section 5.3.10, and it directly modifies this acceleration, helping to avoid numerical instabilites by damping the oscillating velocities.

5.5.4 Storage of Fluids

Another potential bottle-neck for the program comes when looking at how the SPHFluids are stored and passed about. For all calculations, only references are passed to functions needing fluids, which allows for a huge amount of time to be saved when compared to making copies of the Fluid on each function call.

5.6 Collision Detection

This paper also implements a simple interaction of fluids with passive rigid bodies. To do this the passive rigid body is simply represented by points with volumes (spheres) in the same way that Fluid Particles are represented, and these particles are even stored as SPHParticles and SPHFluids. These particles are treated very similiarily to Fluid Particles except that forces do not act on them. A Spatial Hash is performed on all these collision particles, in the same way as with the fluid particles. After finding the new position of all the new fluid particles each of these is tested against its collision neighours(found by querying the collision spatial hash). A very simple collision repsonse is implemented which just multiples the velocity of the particle by a negative scalar $0 \leq \lambda \leq 1$ and moves the particle to the surface of the collision object to avoid inter-penetration. If the passive body is moving at constant velocity this is also taken into account when calculating the velocity change. A similar method is used by Steele et al. (2004).

5.6.1 Calculating Collisions

To calculate the collisions a simple sphere sphere intersection method is used. When setting up passive collision objects the user needs to set up a collision radius. This radius is then used when testing to see whether a fluid particle has penetrated its radius. To do this the squared distance (saves on a square root operation) between the particle and collision particle is calculated. If this distance is greater than the square of the collision radius then there is no collision. However, if this distance is less then there is a pentration. If this is the case then the penetrating particles velocity is made negative and multiplied by the bounce factor and the particle is moved to the collision surface. To move the particle simply involves finding the perentage distance that the particle has penetrated the collision radius by and simply moving it back this distance along its trajectory vector.

5.7 Exporting the simulation

When it comes to visualising the fluid after the simulation has run there is a multitude of methods available. A very common method to use is a combination of Point Splatting and Marching Cube methods (Müller et al., 2003). However, as my main interest is in the simulation itself an existing tool with this functionality is used. In this case Houdini's particle fluid surface is used (Inc., 2010). However, the fact that this is an external application data needs to be exported from the C++ and openGL application to be able to be read and used in Houdini. To do this an animation cache is used and houdini's native file format *.geo*.

5.7.1 Animation Cache

There are two options that arose when deciding how to export data from the program. For both methods, the particle position and velocity need to be exported (for speed stretching in Houdini), for each update. The first option on how to do this is is to write to a file every frame. The problem with this is that this slows down the simulation considerably (as writing to files is a slow operation) especially when dealing with a large number of particles. Another problem is that the simulation may not be of a desirable quality so lots of files would be created that won't end being used.

The second option is the use of an animation cache. Every update an array of fluid positions and velocities is built. This gets added to an animation array containing all the previous arrays so far. When the user is happy with the simulation they can then press a key that will write all of this cache to

individual files (one file per update). This ensures that the simulation is not slowed down by the expensive file writing operations and also that un-wanted simulation data is not written.

In this solver the user gets the option of either method, as the first is very useful for heavy simulations where the computer may run out of memory before the simulation is finished and losing all the cached data if the second option is used. A flow chart of how this works can be seen in Figure 12.

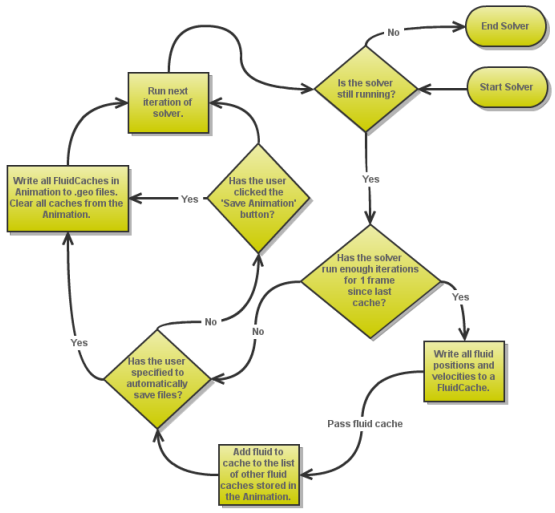


Figure 12: Flowchart representing the caching and exporting process.

5.7.2 Houdini Geo Format

When the users decides to write the cache to a file this needs to be in the *.geo* format which is a readable file format native to houdini. This format is particularly useful as it allows for multiple attributes to be written to a single file. For more information on the *.geo* format please see Inc. (2010).

5.8 Rendering

With the simulation now stored in a sequence of *.geo* files these can be read into houdini using a file SOP. From this point (and velocity) data a 'particle fluid surface' SOP can be attached that creates an implicit surface representing the fluid. The surface size represented by each particle can be controlled by using an attribute called *pscale*. With only these 3 nodes an

accurate representation of the fluid being simulated is created. As seen later in Section 6 this is done automatically with Python Script.

6 Pipeline and usability

As mentioned in 4 the solver that is created needs to be solely controlled from Houdini. The methods implemented to do this will be talked about in the following sections.

6.1 Fluid Digital Asset

As the solver is dependant on these SPHFluid objects an important feature of the pipeline is to allow the user to easily set up multiple fluids in Houdini. Rather than pushing all the fluids into a single node in Houdini, instead a SPHFluid Digital Asset is created.

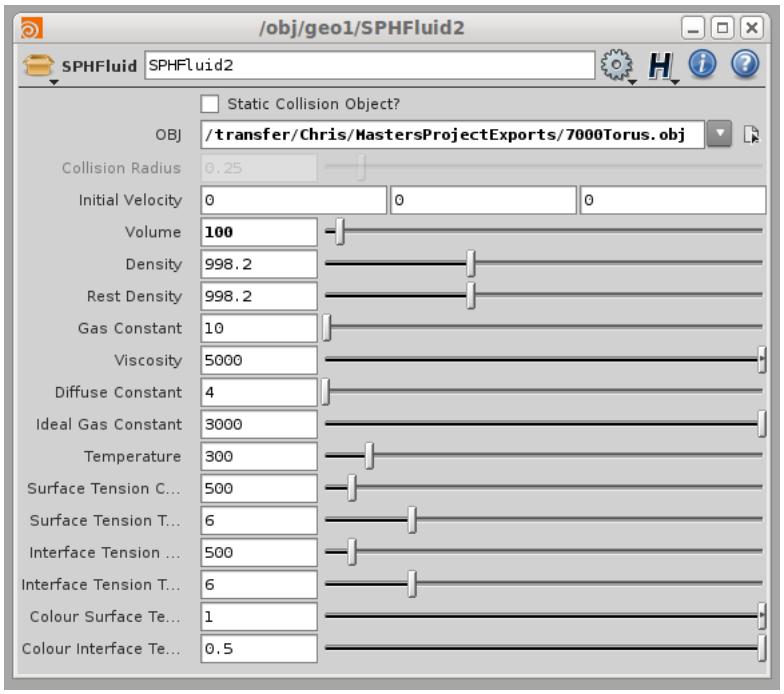


Figure 13: Houdini Custom Digital Asset to control the Fluids.

As you can see in Figure 13 this asset allows for a SPHFluid node to be created with a whole range of parameters to control its behaviour. A toggle switch allows for the fluid to be defined as a static collison object aswell, which removes the unneccesary parameters such as the viscosity and pressure attribues. An important feature of this asset is that multiple fluids can be created quickly, and as seen in the next section, easily plugged into a Solver Asset.

6.1.1 OBJ Loading

In this asset, the fluid that to be created is defined by an OBJ file that needs to be pre-created by the user. For extra clarification on what the fluid will look like when an OBJ is loaded into the asset a Geometry SOP is created with a file node referencing the OBJ inside. This file node is linked to a Copy SOP and Sphere, which pushes a sphere to each particle of the fluid, allowing the user to see more clearly what the fluid will look like. This is done with the use of the HOM and python script, see section 6.3.

6.2 Fluid Solver Digital Asset

Given a number of Fluid Assets, a Solver Asset needs to be created in order to bring them together. This asset needs to control solver specific parameters for the running of the program, the running of the external program and also returning the results back to Houdini. In order to do this a SPHFluidSim asset was created. Inside this asset there are a number of parameters that need to be set to control the workings of the solver itself (such as the location of the C++ binary), see Figure 15 for an example of the asset itself and Figure 14 for how it all fits together.

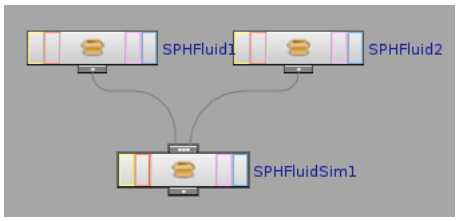


Figure 14: How the two assets are joined in Houdini

With these parameters set, and the fluid assets plugged into the solver, the external solver now needs to be run. This means the information in the assets needs to be transfered from them to the external c++ application somehow. To do this the HOM and python script is used.

6.3 HOM and Python script

The HOM (Houdini Object Model) allows for execution of Houdini functionality using Python script. Inside the Digital Assets in Houdini there is a script tab, which allows for functions to be written that are executed on callback scripts when, for example, buttons inside the asset are pressed. An

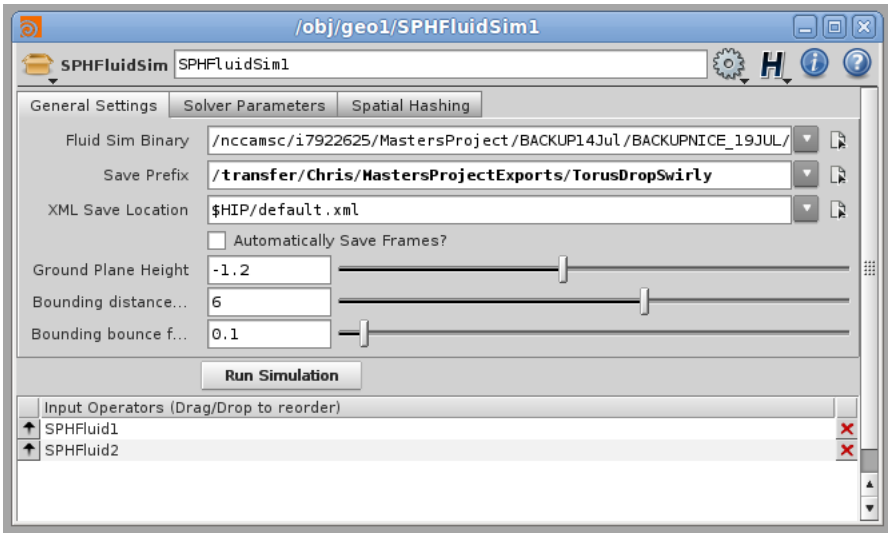


Figure 15: Houdini Custom Digital Asset to control the Solver.

example of this was mentioned earlier, being creating guide geometry when the user chooses an OBJ in the Fluid Asset. The more important use of these script functions come when transfering data from Houdini to the solver application. The method used in this project involves writing the parameter information to an XML file, which can then be read into the C++ Solver application.

6.3.1 XML Configuration Files

6.3.1.1 Writing When looking at getting information from Houdini to a C++ application and back again there is a large number of possible methods. However, the one used in this paper involves writing and reading custom XML files.

Inside the Fluid Solver digital asset there is a button which is called 'Run Simulation'. When this button is pressed a script function inside the asset is run. The first thing that this script does is read through all the parameters in the fluid solver SOP and write these parameters to an XML file under <solver> and <spatialhash> tags. For general layout of the XML files please see Figure 22 in the Appendix.

After it has done this it needs to be able to write the fluid parameters to the file. To do this, for each fluid that is plugged into the solver asset it creates a <fluid> XML block containing all the parameters in the Fluid asset.

After this has been done all the parameters that the C++ application needs to run are contained in a human readable, easy to parse, file format.

After this XML file has been created, the C++ application is run, with the corresponding XML file name passed as an argument, by a simple python command inside the function. This application runs inside a seperate QT window on top of Houdini. Its important to note that even though the user specifies the save location of this file, they never have to interact with it in anyway. It is however the best reference to what parameters a simulation was run with.

6.3.1.2 Parsing The other end of this parameter pipeline involves the C++ application parsing the newly created XML file inside the application, setting up the solver, the spatial hash and the fluids accordingly. This task is relatively trivial due to the XML tags and a collection of if statements tests lines of the file for these tags and sets parameters accordingly.

6.4 Reloading Saved Simulations

During the running of the external C++ application the user has the opportunity to press a button inside the QT Window to save their simulation. When they do this the program writes the simulation so far to a collection of geo files, see Section 5.7.1. As the application is run from the python script inside the fluid solver asset, when it exits it jumps back to this function. Before this function finishes, for extra help to the user any simulation data that has just been written gets brought back into houdini by creating a Simulated Geometry SOP. Inside this SOP contains each fluid as a file node, a createAttribute node (to add the density parameter *pscale*) and a particle fluid surface node, with metaball rendering set, to give the user a true representation of what the simulated fluid looks like.

After this the user can either tweak the current settings of the simulated fluid or re-run the simulation with a whole new range of settings.

7 Results and Analysis

This section looks at examples of some of the results achieved from the solver, along with an analysis on certain features. It also looks at some problems that were faced and solutions to resolve them.

7.1 Example Simulations

Below is a quick look at some of varying results that can be achieved from this solver. It is important to note here that due to the solution found being the correct pyhsical solution, too much movement in any parameter will cause instabilities in the solver. These examples looked at real world values of different fluids and set the parameters accordingly. Please see the accompanying videos to this report for the true effect of the results.

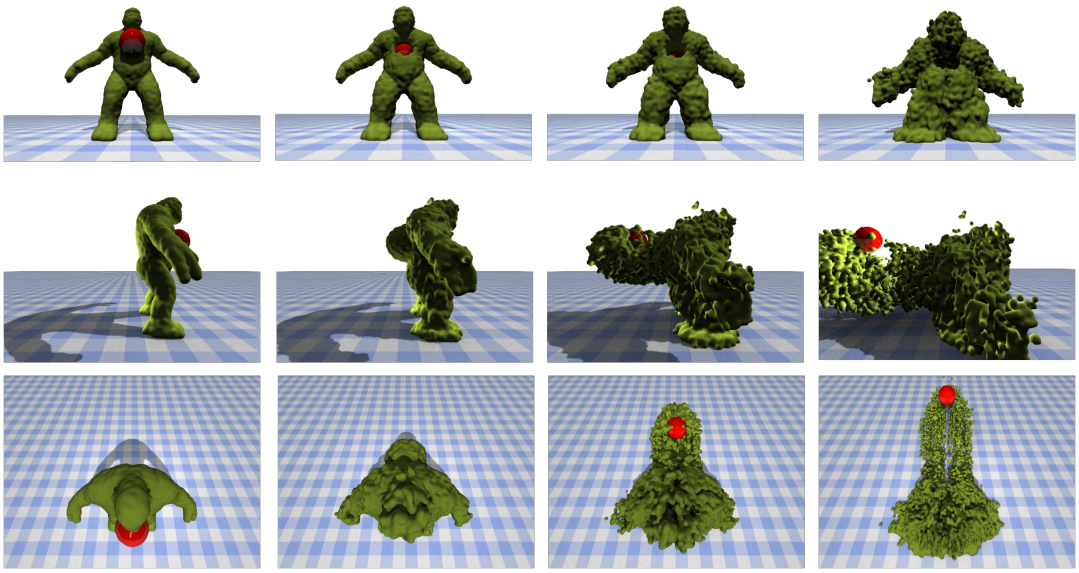


Figure 16: Passive Body and Fluid collision. 40K Particles with medium viscosity (5000) and medium surface tension (500). Model From Ritchie Moore.

In Figure 16 you can see the results achieved when firing a passive rigid body body through a fluid . The passive body forces its way through the fluid causing a hole to be created straight through it. As it leaves the fluid it creates a suitable trail of fluid particles as it would in reality. An interesting effect is how the exit wound is much larger than the entry wound just like in

reality. By representing the passive body in the same way as the fluid this way an easy result to achieve. This figure also represents the ease of creating a fluid from any piece of geometry aslong as it can be saved into an .obj format.

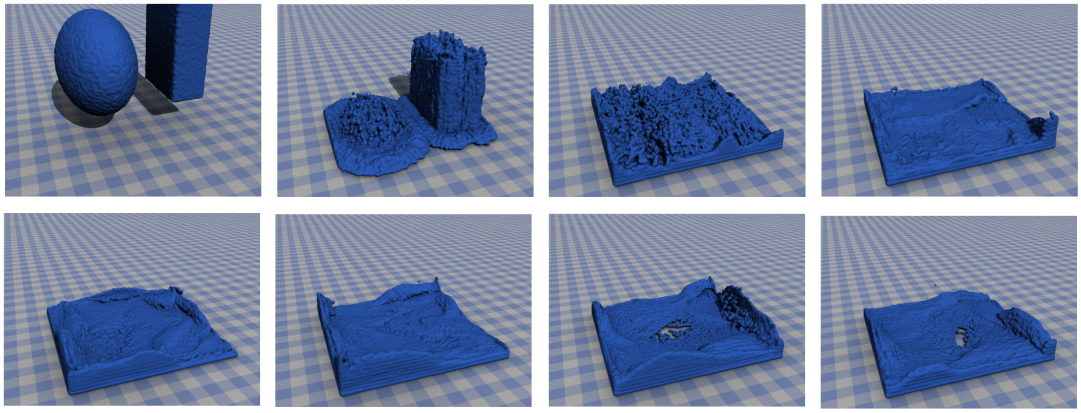


Figure 17: Multiple Fluid Collision. Same polarity fluids mixing together after a drop. Total particle approximately 250K.

The example in Figure 17 looks at the use of multiple fluids in a scene. A large number of particles were used in able to correctly represent the fluid over a large surface area. As you can see in the figure, the two fluids drop and mix together, causing waves and eruptions as they collide. An almost vortex like motion is created with waves crashing against the bounding area. In the video of this simulation, it is easier to see a range of different sized waves created and not just the large ones. This is due to the very high number of particles used, allowing for finer detail to be picked up and simulated.

Figure 18 shows a similiar result but the fluid converges to a single swirling splash in the centre of the bounding area. The results achieved in this example start to show closer resemblance to the behaviour of real fluid.

The results in Figure 19 show how the tool created can be used inside a production. This particle man needed to be blown away and destroyed at the end of Joseph Long's (MADE) effects piece. By just using his OBJ, multiple effects were achieved very quickly allowing the artist to keep tweaking until a perfect simulation was found.

Other examples can also be seen with the accompanying video's, including a look at changing the viscosity of the fluid and also adding temperature.

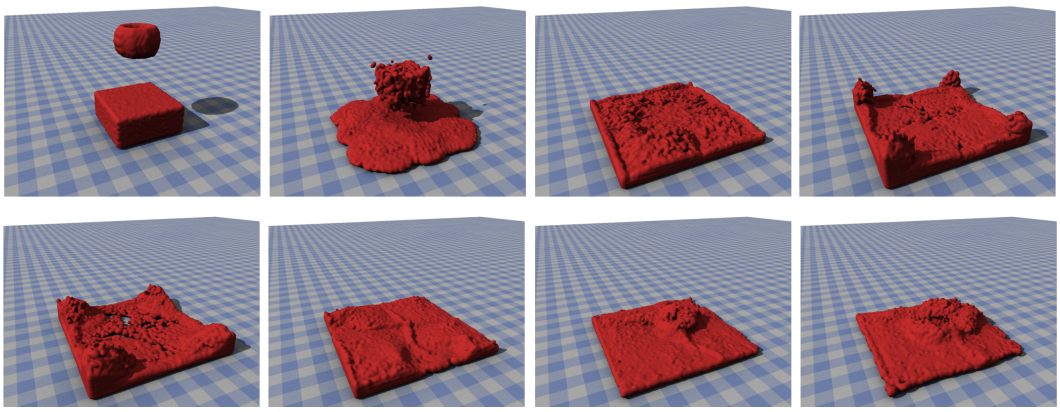


Figure 18: Multiple Fluid Collision. Same polarity fluids mixing together after a drop. Total particle approximately 250K.

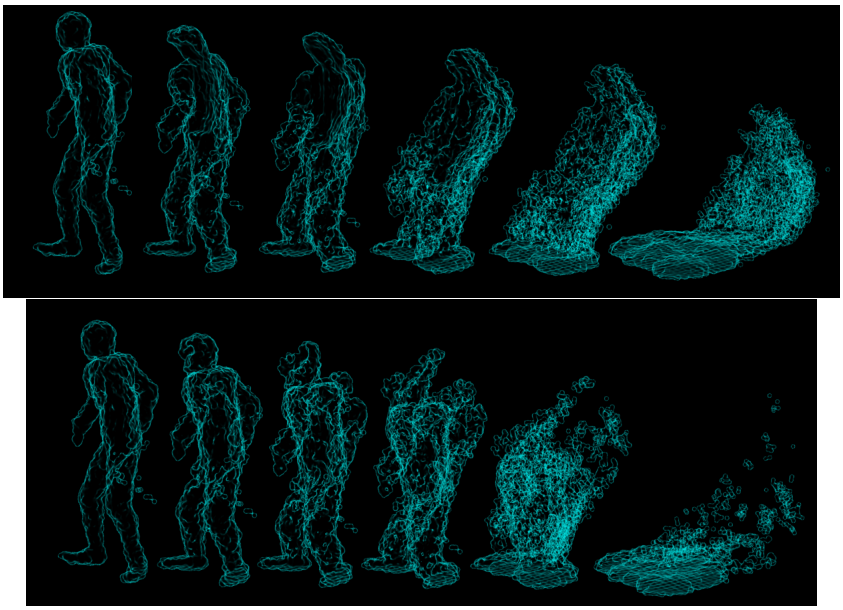


Figure 19: Integration to Real Scene. This is a model by Joseph Long turned into a particle fluid for a dramatic ending to his final effects piece. In both examples collision objects are used to blow holes in the particle man.

7.2 Efficiency

It is hard to quantify the efficiency of this solver due to many parameters controlling simulation time. The most prominent of these parameters is the smoothing length h, the iteration step used in the Spatial Hash and the number of particles. These all directly relate to finding neighbours that then affects how quickly the forces are calculated for each particle. Graph 20 takes a look at how three methods that are the most computationally expensive in the solver. These occur each iteration and are filling the hash table, retrieving neighbours from the hash table and calculating the force equations for all the particles.

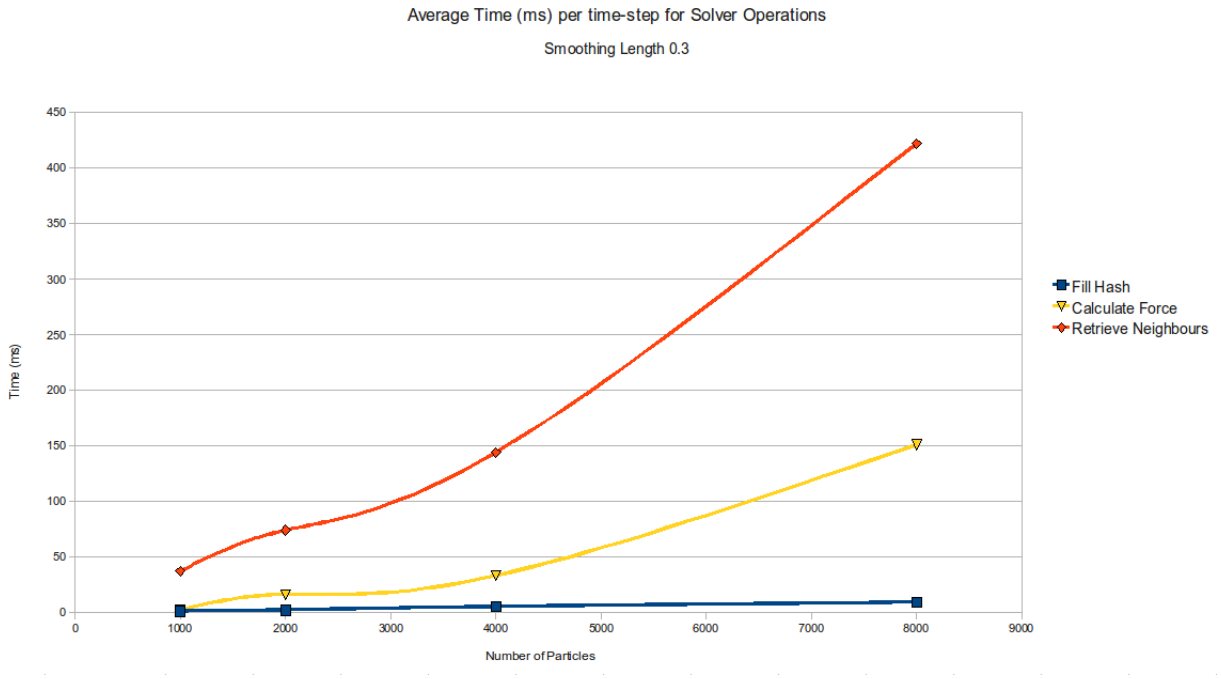


Figure 20: Graph comparing the efficiency of the three most computationally expensive parts of the solver.

In this table you can clearly see that the biggest bottle-neck is when retrieving neighbours. When looked at in more detail it becomes clear that the most expensive operation is creating the key from each particles location. Filling the hash table uses this operation but only once per particle. However, when searching for neighbours this function is used multiple times when iterating across the bounding area, and this occurs for each particle. Obviously, when the smoothing length is increased there are more neighbour

particles so this operation is used more often decreasing the efficiency even further.

This CreateKey() function, although already optimised, is not a computer friendly function. It uses a combination of modulus and floor operations that are expensive operations. Attempts were made at optimising the floor function but different methods proved no quicker than the standard C++ operator.

Another test was run to prove that the Spatial Hashing increased performance compared to a Brute Force method. For this, the same 8000 particle example as above was run. Obviously the neighbour methods are now void but the force calculation took a staggering $40000ms$ (approximate average) to just compute 1 iteration with the same smoothing length. This clearly proves the efficiency increase of using the Spatial Hashing.

7.3 Special Features

When implementing this solver, a new feature was discovered. As mentioned in the implementation, to correctly implement and solve the Navier Stokes equation (3.2.1) the individual pressure's of each particle needs to be updated before the viscosity and pressure force terms can be solved. Using this method gives satisfying results especially for purposes relating to accurate CFD simulations. However, the problem with these simulations, even though physically correct, is that they are relatively boring for visual effects. Instead, in this project a technique was discovered to allow for more exciting simulations with crashing waves.

Instead of calculating all the particle pressure's before the force loop the pressure is updated as and when it is needed to solve the pressure term. Doing this for the density aswell, increases the efficiency of the solver, by removing a whole for loop iterating over all the fluids. In pseudo-code terms what this means is that the first loop in Figure 6 is moved inside the second loop. The most important aspect is the new visual result that is achieved. Fluids now cause crashing waves when they meet each other, with some spectacular effects achieved, see the comparison in Figure 21.

The reason for the new effect is relatively simple. When solving the pressure term, the density and pressure values for the neighbouring particles are actually the values from the previous iteration. By using these values, shifts in the pressure field are created, exaggerating the pressure force between low and high density area's. This exaggeration accentuates the flow of the

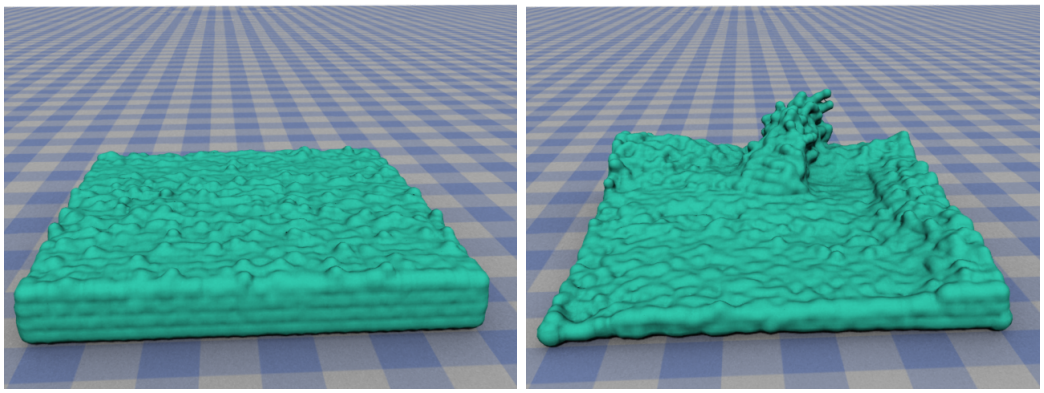


Figure 21: Comparison of Correct Navier Stokes solution (left) and new method (right). The correct solution has settled long before the new method, with the new method crashing colliding parts of the fluids together. These images have exactly the same parameters and the only difference is the solution used to solve Navier Stokes. Both taken at frame 213 of the simulation.

fluid causing more vigourous movement producing more pronounced wave crashes. This effect can almost be compared to vigourous ocean movement rather than fixed fluid simulations. The equation in (7.3.1) is the newly designed equation to solve the pressure term. In this equation the values from the previous iteration are specified by $j-1$.

$$\mathbf{f}_i^{presure} = -\rho_i \sum_{j \neq i} \left(\frac{p_i}{p_i^2} + \frac{p_{j-1}}{p_{j-1}^2} \right) m_j \nabla W_{pressure} \left(\mathbf{x}_i - \mathbf{x}_j, h \right) \tag{7.3.1}$$

This effect really boosts the visual performance of the solver, allowing for more spectacular results to be achieved, more efficiently than before and without extra force calculations. To achieve this affect inside Houdini would take a lot of work and experience in particle fluids (if possible at all) compared to a simple click of a button inside this solver. The user can control whether or not to simulate the forces using the physically correct solution or this new solution, labelled 'Swirly Pressure'.

7.4 Visual accuracy and impact

As mentioned briefly before, this project falls between true physically correct CFD simulations and non-physically correct fluid simulations. For fluids such

as water, the true solution to the Navier Stokes equation (3.2.1) can be used and the results are relatively convincing, although lacking in certain areas compared to advanced CFD sims. This also holds true for viscous materials such as honey but problems do arise here when dealing with very high viscosity fluids, as the viscosity force will start to introduce energy into the scene, causing huge instabilties. However, using the new technique presented in Section 7.3 some spectacular visual results can be achieved. In particular, the new crashing waves add something rarely seen in previous particle fluid simulations, and there exciting nature make them fit right at home in a visual effects sense (where accuracy is not as important as the visual result). The range of effects that this solver can achieve is still relatively undiscovered and in the hands of an effects artist, some even more stunning results could be seen.

7.5 Known Issues

Even with the wide range of successful results achieved there are still a number of issues that this solver presents that need to be taken into account.

7.5.1 Timestep

Choosing the correct timestep is an essential process in getting a visually effective result from this solver. Using Euler's method for integration is completely out of the question, as with it being a low order method the timestep needs to be very low to achieve a result with no instability. Even with the higher order Leap Frog Method, simulations with a high gas constant and viscosity need quite a small timestep (around 0.004) to achieve results that are stable. With the optimisation methods that are implemented this proves to be less of a problem as each step is relatively efficient but preferably a timestep of around 0.01 would be used. Paiva et al. (2009) does approach this problem by using an adaptive timestep, which may solve some of the problems. As this isn't implemented though the best the user can do is run the simulation for a few frames and check its stability, as most simulations will be most unstable at the start.

7.5.2 Smoothing Length

Choosing the correct smoothing length h is also an important process to achieve results that look correct. If this length is too small, the number of neighbours of each particle is too few and the fluid will generally form 1 layer rather than resting on itself. Issues may also arise where particles get really

close to one another and as soon as they breach the smoothing length they are vigourously projected away, causing a simulation that looks like the fluid is boiling.

Choosing a value for h that is too high also causes big problems. The problem occurs that the neighbour particles close to the particle being tested provide less of an influence on that particle than they should as the smoothing kernel has been stretched over a larger distance. Kelager (2006) provides a temporary solution to this by specifiying an average number of particles s that the kernel should support, see equation (7.5.1).

$$h = \sqrt[3]{\frac{3Vs}{4\pi n}} \tag{7.5.1}$$

This however does not really solve the problem as s still needs to be found experimentally. This is almost as much guess work as finding the smoothing length in the first place so in this project it is left to the user to tweak the smoothing length until a suitable result is found.

8 Conclusion

The aim of this study was to design and implement a 3D Particle Fluid Solver that correctly solved the Navier Stokes equations.

In Section 4 a number of criteria were set that this project was designed to meet.

8.1 Multiple Fluid Criteria

The first of these was that multiple fluids should be able to be simulated in the same scene. In this project this criteria is definitely met and examples can be seen in section 7.1.

Multiple fluids are correctly simulated individually and interact accordingly, updating all forces and even fluid/fluid forces such as Surface Interface Tension. With the Spatial Hashing technique this fluid/fluid interaction is very efficient with multiple fluids that are far away not being tested against each other until they get closer together.

8.2 Extendable Criteria

The second of the criteria's that needed to be met was that the project and framework needed to be easily extendable. This is proven to be met by looking at the number of parameters and huge range of effects that can be achieved. In the SPHForce class you are able to specify whether certain forces are to be calculated or not such as the Temperature force, allowing for simulations to be optimised depending on the user needs. As you can see from the video's or some of the images a very wide range of effects can be achieved due to the large number of customisable parameters, ranging from crashing waves to viscus bunny rabbits. Methods such as artificial viscosity and XSPH also extend the functionality of the solver allowing for different effects to be achieved.

From a developers point of view the Framework is very extendable as the methods are written in such a way that they can create their own similiar methods to meet there needs. This is made easier by the heavy use of object orientation, allowing the developer to easily find and tweak individual methods, such as an Integration method, without impacting on the other components of the program. By this nature of the project, a developer could also pick up this framework and be able to extend and add new features to design any other particle based methods, such as a cloth simulation or an advanced particle system, without having to rewrite code.

8.3 Collisions Criteria

Criteria three of the design stated that the simulation should allow for simple interaction with Static Rigid Bodies and bounding areas. This project satisfies this criteria with the modelling of static bodies as fluids (with no update of forces) allowing for the re-using of the spatial hash methods and easy testing for possible neighbours. This means that multiple collision objects, of any form, can be added to the scene, and with the introduction of the collison radius parameter and bounce factor, interact successfully with each other. Allowing the static bodies to also have velocity effects as a bullet passing through a liquid or a ball being dropped into a bowl of water can be achieved.

8.4 Efficiency Criteria

The fourth criteria that was set states that the solver must be efficient in passing large data structures around and interactions between particles must be optimised. The first part of this criteria is met by the use of only passing fluids by reference. This greatly increases the speed of the simulation as, especially with fluids containing a large number of particles, every fluid is not copied when it is passed to a function. Instead, only 1 instance of a fluid is created and is simply passed the reference to where this fluid is stored in memory and updates the memory address accordingly.

The second part of this criteria was met with the Spatial Hashing Technique. This technique allowed for the removal of the brute force method (testing every particle against every other particle) and the Spatial hashing technique introduced a method that efficiently found neighbours reducing the computational time dramatically, see Section 7.2

8.5 Further work

With only limited time to implement features there still exists a number of them that can easily fit into the current framework with future work.

8.5.1 Dynamic Air Particles

Müller et al. (2005) presents a very nice method for the introduction of air particles to the simulation. Normally to simulate air you would have to simulate all of the air particles, which is unfeasible for large scenes. Instead, the method presented looks at the gradient of the particles surface (in a similiar fashion to finding the surface tension) and dynamically creates particles

where air would get trapped in the simulation. The air particles have a artificial buoyancy force which lifts them up through the fluid until they leave the fluid, where they are then destroyed. This is relatively simple to implement and would easily fit into the current framework.

8.5.2 Visco-Plastic and Non-Newtonian Fluids

Even though this solver can produce viscous fluids, it does not correctly implement non-newtonian fluid effects. There are multiple papers that look at solving this, with Paiva et al. (2009)'s paper currently implementing a very effective, artist friendly solution. Non-newtonian fluids notouriously involve a large number of parameters and Paiva's use of a jump number simplifies a multitude of parameters into a single one. This solution would also be easily implemented into the current framework, with multiple features already implemented such as the artificial viscosity and XSPH Particle correction.

8.5.3 Interaction with Full Rigid Body System

An area where this solver lacks compared to Houdini's native solver is in its relationship with a full Rigid Body System. Currently, the solver presented in this paper can only accept passive rigid bodies that don't receive forces back from the fluid. There are multiple options available in order to do this, including writing your own Rigid Body Solver yet this is a big project in itself so would need a considerable amount of work. Another option of course would be to integrate an existing solver such as 'Bullet', which would take control of all the Rigid Body side of things (Library, 2010).

8.5.4 Multi-threading and GPU

Another area that would help boost the performance of this solver dramatically would be the use of GPU programming. Many papers exist that explore this as, by the nature of the solver, the theory behind doing this is realtively simple. Each particle of the fluid is having the same instructions performed on it, and do not depend on each other. Therefore, in theory each particles instructions could be carried out at the same time, improving the simulation x times where x is the number of machines/cores available to solve the force equations.

References

Batchelor, G. (1973). An introduction to fluid dynamics.

Bridson, R., Fedkiw, R. and Muller-Fischer, M. (2006). Fluid simulation: Siggraph 2006 course notes fedkiw and muller-fischer presenation videos are available from the citation page, *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, pp. 1–87.

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

Chang, Y., Bao, K., Liu, Y., Zhu, J. and Wu, E. (2009). A particle-based method for viscoelastic fluids animation, *VRST '09: Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*, ACM, New York, NY, USA, pp. 111–117.

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

Desbrun, M. and Gascuel, M.-P. (1996). Smoothed particles: A new paradigm for animating highly deformable bodies, *In Computer Animation and Simulation 96 (Proceedings of EG Workshop on Animation and Simulation*, Springer-Verlag, pp. 61–76.

Fedkiw, R., Stam, J. and Jensen, H. (2001). Visual simulation of smoke., *In SIGGRAPH 2001 Conference Proceedings, Annual Conference Series*, pp. 15–22.

Fletcher, C. (1990). Computational techniques for computer graphics., *ACM Transactions on Graphics 1*.

Foster, N. and Dimitri, M. (1996). Realistic animation of liquids.

Harris, M. (2004). Fast fluid dynamics simulation on the gpu - gpu gems series, `http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html`. First accessed: 10th April.

Inc., S. S. (2010). Sidefx houdini 3d animation tools, `http://www.sidefx.com/`.

Kaw, A. (2010). Instruction videos for numerical methods, `http://numericalmethods.eng.usf.edu/videos/`. First accessed: 29th May 2010.

Kelager, M. (2006). Lagrangian fluid dynamics using smoothed particle hydrodynamics.

Koshizuka, S., Tamako, H. and Oka, Y. (1995). A particle method for incompressible viscous flow with fluid fragmentation, *Computational Fluid Dynamics Journal* **4**: 29–46.

Lamb, H. (1994). Hydrodynamics (6th edition).

Library, B. P. (2010). Bullet physics library, `http://bulletphysics.org/wordpress/`.

Macey, J. (2010). Ngl graphics library, `http://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/index.html`.

Mao, H. (2006). *Physical-based non-newtonian fluid animation using sph*, PhD thesis, Edmonton, Alta., Canada.

Mao, H. and Yang, Y.-H. (2006). Particle-based immiscible fluid-fluid collision, *GI '06: Proceedings of Graphics Interface 2006*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, pp. 49–55.

Monaghan, J. (1992). Smoothed particle hydrodynamics, *Annual Review of Astronomy and Astrophysics*, pp. 543–574.

Monaghan, J. J. (1989a). On the problem of penetration in particle methods, *J. Comput. Phys.* **82**(1): 1–15.

Monaghan, J. J. (1989b). On the problem of penetration in particle methods, *J. Comput. Phys.* **82**(1): 1–15.

Müller, M., Charypar, D. and Gross, M. (2003). Particle-based fluid simulation for interactive applications, *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 154–159.

Müller, M., Solenthaler, B., Keiser, R. and Gross, M. (2005). Particle-based fluid-fluid interaction, *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, pp. 237–244.

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

Priscott, C. (2010). Cgi techniques, interactive 2d fluid simulation.

Stam, J. (1999). Stable fluids., *Conference Proceedings, Annual Conference Series*.

Stam, J. (2001). A simple fluid solver based on the fft., *Journal of Graphics Tools*, Vol. 6.

Stam, J. (2003). Real-time fluid dynamics for games., *Proceedings of the Game Developers Conference*.

Steele, K., Cline, D., Egbert, P. K. and Dinerstein, J. (2004). Modeling and rendering viscous liquids: Research articles, *Comput. Animat. Virtual Worlds* **15**(3-4): 183–192.

Stora, D., Agliati, P.-O., Cani, M.-P., Neyret, F. and Gascuel, J.-D. (1999). Animating lava flows, *Proceedings of the 1999 conference on Graphics interface '99*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 203–210.

Teschner, M., Heidelberger, B., Mueller, M., Pomeranets, D. and Gross, M. (2003). Optimized spatial hashing for collision detection of deformable objects, pp. 47–54.

Win, M. (2007). The implementation of 2d fluid solver plug-in for houdini 8.0.

9 Appendix

9.1 XML Layout

```
<solver>
  <XMLLocation>/nccamsc/i7922625/MastersProject/BACKUP14Jul/BACKUPNICE_19JUL/default2.xml</XMLLocation>
  <m_smoothingLength>0.300000011921</m_smoothingLength>
  <GroundPlaneHeight>-1.20000004768</GroundPlaneHeight>
  <BoundingDistance>7.0</BoundingDistance>
  <BoundingBounce>0.10000000149</BoundingBounce>
  <AutomaticSave>1</AutomaticSave>
  <m_timestep>0.00400000018999</m_timestep>
  <SavePrefix>/transfer/Chris/MastersExports/SphereBoxSwirlyV2</SavePrefix>
  <TemperatureToggle>0</TemperatureToggle>
  <SwirlyPressureToggle>1</SwirlyPressureToggle>
  <m_staticCollisionBounce>0.5</m_staticCollisionBounce>
</solver>

<spatialhash>
  <m_iterateStepFluid>0.899999976158</m_iterateStepFluid>
  <m_iterateStepStatic>0.899999976158</m_iterateStepStatic>
</spatialhash>

<fluid>
  <LoadOBJPath>/nccamsc/i7922625/MastersProject/BACKUP14Jul/BACKUPNICE_19JUL/118000LongBox.obj</LoadOBJPath>
  <m_initialVelocity>0.0 -5.0 0.0</m_initialVelocity>
  <StaticObject>0</StaticObject>
  <m_volume>1000.0</m_volume>
  <m_density>998.200012207</m_density>
  <m_restDensity>998.200012207</m_restDensity>
  <m_gasConstant>10.0</m_gasConstant>
  <m_viscosity>3000.0</m_viscosity>
  <m_colourSurfaceTension>1.0</m_colourSurfaceTension>
  <m_colourInterfaceTension>0.5</m_colourInterfaceTension>
  <m_diffuseConstant>4.0</m_diffuseConstant>
  <m_idealGasConstant>3000.0</m_idealGasConstant>
  <m_temperature>300.0</m_temperature>
  <m_surfaceTensionCoeff>500.0</m_surfaceTensionCoeff>
  <m_interfaceTensionCoeff>10.0</m_interfaceTensionCoeff>
  <m_surfaceTensionThreshold>6.0</m_surfaceTensionThreshold>
  <m_interfaceTensionThreshold>6.0</m_interfaceTensionThreshold>
</fluid>
```

Figure 22: Example XML file, written by Houdini and parsed by the C++ application.

9.2 Spatial Hash Iteration Code

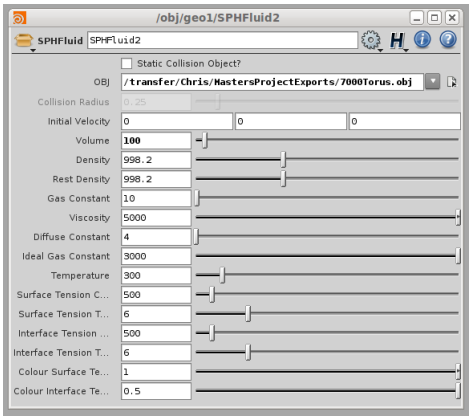
```
// Vector to store the new Key
ngl::Vector newKey;

for(ngl::Real i = BMinX; i < BMaxX; i = i + m_iterateStep)
  {
    for(ngl::Real j = BMinY; j < BMaxY; j = j + m_iterateStep)
    {
      for(ngl::Real k = BMinZ; k < BMaxZ; k = k + m_iterateStep)
      {
        newKey.Set(i,j,k,0);
        // We now call the CreateKey Function which hashes our position for us
        testKey = CreateKey(newKey);
        // We then ask for all the neighbours in this cell
        ppp = m_hashNeighbours.equal_range(testKey);
        // For each neighbour we test to see whether the
        // neighbour is already in the list.
        for (std::multimap<int,SPHParticle>::iterator it2 = ppp.first;it2 != ppp.second;++it2)
        {
          inList = false;
          for(unsigned int q = 0; q < neighbours.size(); ++q)
          {
            if((*it2).second.GetID() == neighbours[q].GetID())
            {
              // If we find that the neighbour is already
              // in the list we break out of this iteration
              inList = true;
              break;
            }
            else
            {
            }
          }
          if(inList == false)
          {
            // If neighbour is unique add to List
            neighbours.push_back((*it2).second);
          }
        }
      }
    }
  }
```

Figure 23: Iteration step of Spatial Hash

9.3 Short User Guide

SPHFluid Asset:

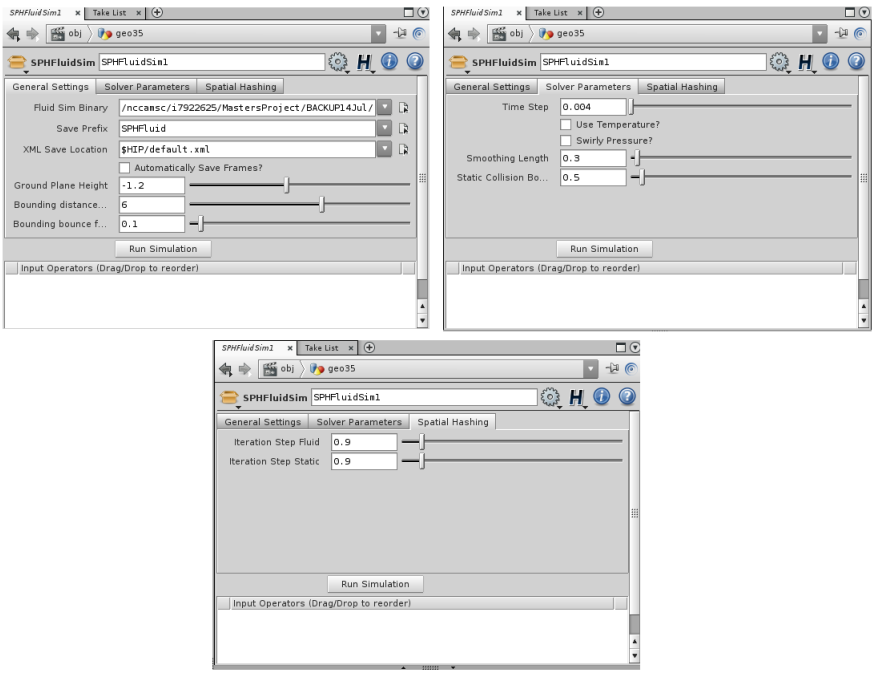


The figure above is an example of an SPHFluid asset. Below runs through a brief description of each parameter:

- Static Collision Object Toggle - If this is set then all the parameters other than OBJ, collision radius and initial velocity will disappear. The fluid is now configured as a collision object rather than a fluid so won't deform shape across the simulation or be affected any forces.
- OBJ - This specifies the location of the OBJ to turn into a fluid. A fluid particle will placed at each vertex of this OBJ.
- Initial Velocity Sets the fluid to have this initial velocity.
- Volume The volume of the fluid to be simulated. If the simulation is blowing up try and raising or lowering this value until the fluid is still settled after a few frames of iteration.
- Density The density of the fluid.
- Rest Density The rest density of the fluid. Used to calculate the pressure. Constant through out the simulation. Set to the default for water.
- Gas constant Constant used when calculating the pressure. The higher the value, the more gaseous the fluid.

- Viscosity Determines how viscous the fluid is. Useful as a damping force to the pressure although if pushed too high then energy is introduced into the scene causing obvious problems.
- Diffuse Constant Determines how quickly the temperature diffuses across the fluid.
- Ideal Gas Constant Determines how much the temperature affects the rest density.
- Temperature The temperature of the fluid in celcius.
- Surface Tension Coefficient Determines how strongly the fluid holds its surface.
- Surface Tension Threshold Controls the gradient at which the fluid holds its surface.
- Interface Tension Coefficient and Threshold - Same as Surface tension but controls tension between multiple fluids.
- Colour Interface Tension Controls the polarity of the fluid. Should be set to -0.5 for a polar fluid and 0.5 for a polar fluid.

SPHSolver Asset:



The figures show an example of an SPHSolver asset. Below runs through a brief description of each parameter:

- Fluid Sim Binary - The location of the C++ binary. Take care when changing this.
- Save Prefix - The location to save the file followed by the prefix to the frame names.
- XML Save Location - The location to save the XML file and its name1.
- Automatic Save Toggle - Turning this toggle on means that everytime the cache has received enough data for a frame it will save out a .geo file of that frame, clearing the cache afterwards. If it is not toggle on then the user needs to press the save simulation button in the QT window to save the cache so far. Note, do not press the save simulation button of automatic save is on as this may cause frame number errors.
- Ground Plane Height - The height of the ground plane.
- Bounding distance - The distance of the bounding walls in x and z from 0,0,0.

- Bounding bounce factor - The bounce factor when a fluid particle hits either the ground plane or boundin walls.
- TimeStep - The timestep to perform each iteration of the solver over. 0.004 is a good value for most simulations.
- Use Temperature toggle - Performs temperature calculations.
- Use Swirly Pressure - Implements new technique for solving the pressure force. If untoggled, the correct Navier Stokes solution is used.
- Smoothing Length - The length at which particles will start affecting each other.
- Static Collision Bounce - The bounce factor when a fluid hits a passive collision object.
- Iteration Step Fluid - The iteration step to take over the bounding area when finding neighbours of the fluid particles.
- Iteration Step Static - The same as above except for the collision particle rather than the fluid particles.
- Run Simulation - Runs the simulation, writing all parameters to the xml file and creating fluid surfaces when the simulation stops.

Note that the solver asset needs fluid assets plugged in to work.

9.4 Houdini Solver Python Script

Below is the script that is run when the 'Run Simulation' button is pressed.

```
import os

def Hello():

    hipExpand = hou.expandString('$HIP')
    homeExpand = hou.expandString('$HOME')
    print hipExpand

    var = hou.evalParm("xml_location")
    var.replace("$HIP", hipExpand)
    var.replace("$HOME", homeExpand)
    xmlLoc = var

    filename = var
    FILE = open(filename,"w")

    #WRITE SOLVER PARAMS TO XML

    FILE.write("<solver>\n")
    FILE.write("\t<XMLLocation>")
    FILE.write(str(var) + "</XMLLocation>\n")

    var = hou.evalParm("m_smoothing_length")
    FILE.write("\t<m_smoothingLength>")
    FILE.write(str(var) + "</m_smoothingLength>\n")

    var = hou.evalParm("ground_plane_height")
    FILE.write("\t<GroundPlaneHeight>")
    FILE.write(str(var) + "</GroundPlaneHeight>\n")

    var = hou.evalParm("bounding_distance")
    FILE.write("\t<BoundingDistance>")
    FILE.write(str(var) + "</BoundingDistance>\n")

    var = hou.evalParm("bounding_bounce")
    FILE.write("\t<BoundingBounce>")
    FILE.write(str(var) + "</BoundingBounce>\n")

    var = hou.evalParm("automatic_save")
    FILE.write("\t<AutomaticSave>")
    FILE.write(str(var) + "</AutomaticSave>\n")

    var = hou.evalParm("m_timeStep")
    FILE.write("\t<m_timestep>")
    FILE.write(str(var) + "</m_timestep>\n")

    var = hou.evalParm("save_prefix")
    var.replace("$HIP", hipExpand)
    var.replace("$HOME", homeExpand)
    FILE.write("\t<SavePrefix>")
    FILE.write(str(var) + "</SavePrefix>\n")


    var = hou.evalParm("temp_toggle")
    FILE.write("\t<TemperatureToggle>")
    FILE.write(str(var) + "</TemperatureToggle>\n")
    var = hou.evalParm("swirly_toggle")
```

```
FILE.write("\t<SwirlyPressureToggle>")
FILE.write(str(var) + "</SwirlyPressureToggle>\n")

var = hou.evalParm("collision_bounce")
FILE.write("\t<m_staticCollisionBounce>")
FILE.write(str(var) + "</m_staticCollisionBounce>\n")

FILE.write("</solver>\n\n")

#WRITE SPATIAL HASH PARAMS TO XML

FILE.write("<spatialhash>\n")
var = hou.evalParm("m_iterate_step_fluid")
FILE.write("\t<m_iterateStepFluid>")
FILE.write(str(var) + "</m_iterateStepFluid>\n")

var = hou.evalParm("m_iterate_step_static")
FILE.write("\t<m_iterateStepStatic>")
FILE.write(str(var) + "</m_iterateStepStatic>\n")

FILE.write("</spatialhash>\n\n")

#WRITE FLUID PARAMS TO XML

currentNode =  hou.Node.path(hou.pwd())
me = hou.node(currentNode)
print hou.Node.inputs(me)
inputs = hou.Node.inputs(me)
numFluids = 0

for fluid in inputs:
    FILE.write("<fluid>\n")
    numFluids = numFluids + 1
    #For the path we need to strip the start

    var = fluid.evalParm("obj")
    var.replace("$HIP", hipExpand)
    var.replace("$HOME", homeExpand)

    FILE.write("\t<LoadOBJPath>")
    FILE.write(str(var) + "</LoadOBJPath>\n")

    var = fluid.evalParm("initial_velocityx")
    strvar = str(var) + "_" + str(fluid.evalParm("initial_velocityy"))
    strvar = strvar  + "_" + str(fluid.evalParm("initial_velocityz"))
    FILE.write("\t<m_initialVelocity>")
    FILE.write(strvar + "</m_initialVelocity>\n")

    var = fluid.evalParm("collision_toggle")
    FILE.write("\t<StaticObject>")
    FILE.write(str(var) + "</StaticObject>\n")



    if var == 1:
        var = fluid.evalParm("collision_radius")
        FILE.write("\t<m_collisionRadius>")
        FILE.write(str(var) + "</m_collisionRadius>\n")
    elif var == 0:
```

```
    var = fluid.evalParm("m_volume")
    FILE.write("\t<m_volume>")
    FILE.write(str(var) + "</m_volume>\n")

    var = fluid.evalParm("m_density")
    FILE.write("\t<m_density>")
    FILE.write(str(var) + "</m_density>\n")

    var = fluid.evalParm("m_restDensity")
    FILE.write("\t<m_restDensity>")
    FILE.write(str(var) + "</m_restDensity>\n")

    var = fluid.evalParm("m_gasConstant")
    FILE.write("\t<m_gasConstant>")
    FILE.write(str(var) + "</m_gasConstant>\n")

    var = fluid.evalParm("m_viscosity")
    FILE.write("\t<m_viscosity>")
    FILE.write(str(var) + "</m_viscosity>\n")

    var = fluid.evalParm("m_colourSurfaceTension")
    FILE.write("\t<m_colourSurfaceTension>")
    FILE.write(str(var) + "</m_colourSurfaceTension>\n")

    var = fluid.evalParm("m_colourInterfaceTension")
    FILE.write("\t<m_colourInterfaceTension>")
    FILE.write(str(var) + "</m_colourInterfaceTension>\n")

    var = fluid.evalParm("m_diffuseConstant")
    FILE.write("\t<m_diffuseConstant>")
    FILE.write(str(var) + "</m_diffuseConstant>\n")

    var = fluid.evalParm("m_idealGasConstant")
    FILE.write("\t<m_idealGasConstant>")
    FILE.write(str(var) + "</m_idealGasConstant>\n")

    var = fluid.evalParm("m_temperature")
    FILE.write("\t<m_temperature>")
    FILE.write(str(var) + "</m_temperature>\n")

    var = fluid.evalParm("surface_tension_coeff")
    FILE.write("\t<m_surfaceTensionCoeff>")
    FILE.write(str(var) + "</m_surfaceTensionCoeff>\n")

    var = fluid.evalParm("interface_tension_coeff")
    FILE.write("\t<m_interfaceTensionCoeff>")
    FILE.write(str(var) + "</m_interfaceTensionCoeff>\n")

    var = fluid.evalParm("surface_tension_threshold")
    FILE.write("\t<m_surfaceTensionThreshold>")
    FILE.write(str(var) + "</m_surfaceTensionThreshold>\n")

    var = fluid.evalParm("interface_tension_threshold")
    FILE.write("\t<m_interfaceTensionThreshold>")
    FILE.write(str(var) + "</m_interfaceTensionThreshold>\n")


FILE.write("</fluid>\n\n")
print "Gets_in_loop"
print fluid.evalParm("m_volume")
```

```
FILE.close()

#Run the Program

binary = hou.evalParm("binary")
os.system(binary + "␣" + xmlLoc)

path = hou.evalParm("save_prefix")
path = path + "_Fluid"
home = hou.node('/obj')
simGeo = home.createNode('geo', "Simulated_Fluid")
children = simGeo.children()
children[0].destroy()
merge = simGeo.createNode('merge')
for i in range(numFluids):
    fileNodeName = "Fluid_" + str(i)
    #print fileNodeName
    fileNode = simGeo.createNode("file", fileNodeName)
    newPath = path + str(i) + "_Frame_$F.geo"
    fileNode.parm("file").set(newPath)
    fileNode.moveToGoodPosition()
    attrNode = simGeo.createNode("attribcreate")
    attrNode.parm("name").set("pscale")
    attrNode.parm("value1").set(0.3)
    attrNode.setNextInput(fileNode)
    attrNode.moveToGoodPosition()
    fluidSurface = simGeo.createNode("particlefluidsurface")
    fluidSurface.setNextInput(attrNode)
    fluidSurface.moveToGoodPosition()
    fluidSurface.parm("method").set("metaball")
    merge.setNextInput(fluidSurface)

merge.moveToGoodPosition()
merge.setDisplayFlag(True)
merge.setRenderFlag(True)
```