Master Thesis

Simulation and Rendering of a Viscous Fluid using Smoothed Particle Hydrodynamics

Tutor: Kenneth Holmlund

Author: Marcus Vesterlund
(http://mzlair.se)

3rd December 2004

In memory of my mother, Lisbeth Vesterlund (1944-2002), and my brother, Lars Vesterlund (1966-2001).

Abstract

In this master thesis, a system for simulating and rendering a viscous fluid in real-time, using particles, is implemented. Mesh-free methods are gaining new ground and are also used through out this system. The simulation uses a relatively new method within computer graphics, called Smoothed Particle Hydrodynamics. A new and effective method for extracting a surface from the simulation, for rendering using point splatting, is presented. An effective implementation in real time is achieved with plausible results.

Contents

1 Introduction

In everyday life we usually do not think of how complex ordinary fluids (i.e. liquids and gases) behave. The raising smoke from a camp fire. The breaking of an ocean wave when cutting a tube[1]. Turbulence in miniature when stirring a cup of Latte. The list ordinary fluid phenomenas can go on, but, however ordinary they may seem, they present a tough computational problem.

The field of research on this problem is called Computational Fluid Dynamics (CFD). It has been well established for a long time, but there still exists many loose ends. Typical applications for computational fluids lay within the field of engineering, with greater demands on being accurate then on the computational costs. However, Moore's law and the use of less accurate methods are beginning to enable more and more complex real-time simulations. Real-time implementations might make their way into computer games and other virtual environments. A real-time implementation can also be used to previewing a more accurate non real-time simulation.

We will use a method called Smoothed Particle Hydrodynamics (SPH) for achieving real-time frame-rates. The SPH method utilizes a novel and innovative form of local field approximation to differentials and functions making it possible to solve a differential field equation using an intuitive and extensible particle method. One could see it as if the particles span a dynamic mesh which is used to discretize and solve the continuum equations, but there is actually more to the method than this. The core of the method is the use of a smoothing kernel, i.e. a function with certain properties that is used to sum up an approximate value of the field functions from the surrounding particles.

The SPH method is what we call a mesh-free method. Mesh-free methods are generally gaining in popularity since they are both computationally and numerically efficient, and also easy to implement and use. Applications lie in e.g. simulation of fluids (water, blood, etc), soft matter (e.g. skin, organs) and granular materials (e.g. sand and mud). All of these applications have considerable industrial impact too, e.g. in the game industry, movie special effects, medical training simulators, vehicle simulators, etc. Possibly the impact will be quite big on future computer games - imagine a character that can change it's form and structure from solid to fluid a´la Terminator

[1] To succeed surfing within a breaking wave.

2 ([CJ91]).

Modern GPUs are becoming more and more powerful. They have more transistors than a CPU. Real-time visualization of complex materials requires GPU shader programs to be used to take the load off the CPU. The CPU has better uses, for example physical simulation or AI in computer games. The GPU can also be used for the physical simulation (see [GPG] and [AIY+03] for example) as it has powerful parallel processing capabilities, but this is beyond the scope of this thesis.

Point-based surface representations are on the march. It has been established as a viable graphics rendering primitive in [Gro01]. And high-quality texture sampling and rendering have been developed in [ZPBG01]. Lots of other articles have been written, for a good survey see [Jar03]. It is also a mesh-free method, which conceptually fits with using the SPH method.

In the next chapter we will take a quick look at the objectives. In the following chapter we will have a look at possible existing algorithms and methods used to solve the problems involved with the simulation and rendering. This will include mathematical, physical and computationally aspects. When we are done with existing algorithms and methods we will see how they are used in the implementation. Not all of the presented methods are used. Some new methods will be presented. This is followed by results and then conclusions.

2 Objective

The main objectives of this thesis was to:

1. Implement a general particle system. This system should specifically be able to handle internal forces between the particles.
2. Implement a SPH simulation using this particle system.
3. Implement and develop rendering of this simulation.

These implementations should be for real-time or near real-time execution. It should be implemented for standard desktop computers with the latest graphics hardware and an easily extendable and effective API should be created. Preferably OpenSceneGraph ([Ope]) would be used for rendering.

3 Overview of methods

In this chapter we will survey methods needed for efficient simulation of viscous fluids with the SPH methods and for rendering the results.

3.1 Smoothed Particle Hydrodynamics

The Smoothed Particle Hydrodynamics (SPH) method was originally developed in 1977 for simulating flow of interstellar gas within Astrophysics. It was developed by Bob Gingold and Joe Monaghan [GM77] and independently by Leon Lucy [Luc77]. The SPH method is a powerful particle method for the solution of complex fluid dynamical and material problems. It has, since it was developed, been used successfully in a wide range of areas, including shock wave phenomena, breaking waves, lava flow, elastic-fracture, free-surface problems and granular matter. It is sometimes presented as a next generation method over finite element analysis, with a predicted strong impact on the entire field of computation [LL03].

We will now take a look at the mathematics and physics behind the SPH simulation. Beginning with the SPH mathematics and then moving on to physical theory and how it applies to the SPH model. This is only a quick survey over what is needed for this thesis. For a more thorough introduction and overview, please read [LL03].

3.1.1 SPH basics

There are two conceptual different methods to describe fluid dynamics, the Eulerian method and the Lagrangian method [Whi94]. The Eulerian method uses spatial coordinates and the Lagrangian method uses material coordinates. To visualize this, the first method can be seen as fixing a grid on top of the fluid and, at each time step, observe the velocities at the grid points. The second one can be seen as having corks floating along with the fluid and, at each time step, observe the positions of the corks.

There are also mesh-based methods and mesh-less methods. The difference is the explicit connectivity information carried by the mesh based method. Eulerian methods are almost always mesh based while Lagrangian methods have implementations in both mesh-based methods and mesh-less methods.

One drawback of using Lagrangian mesh-based methods is that can cause deformation of the mesh. A deformed mesh can cause bad conditioning of the simulated problem.

The SPH method is a purely mesh-less Lagrangian method. Most commonly used for fluid simulations is the Finite Element Method, or FEM for short. FEM is almost the exact opposite of SPH as FEM is a mesh-based Eulerian method. Most common liquid simulation for real-time applications is simulating ripples on a surface using a mesh-based Lagrangian method. This is done by modeling the surface as springs and particles.

The SPH simulation technique is based on interpolation theory. The basic interpolation formula used is,

$$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \; . \tag{1}$$

This equation describes how a scalar value A_S is interpolated from sampled points A_j.

Function W is called the kernel function, or the smoothing kernel. The kernel function has a cut off radius, h. The cut off radius sets $W = 0$ for $|\mathbf{r} - \mathbf{r}_j| > h$. W should also be differentiable for many applications and the following properties must hold,

$$\int_{\mathbf{r}} W(\mathbf{r}, h) d\mathbf{r} = 1 \tag{2}$$

and

$$\lim_{h \to 0} W(\mathbf{r}, h) = \delta(\mathbf{r}) \; . \tag{3}$$

By trying to interpolate the density in equation 1 we retain the density field,

$$\begin{aligned} \rho_S(\mathbf{r}) &= \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \\ &= \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \; . \end{aligned} \tag{4}$$

The density field is sampled at the the location of the particles, $\rho_j = \rho_S(\mathbf{r}_j)$.

The derivates are often used in most fluid equations. And now we come to one of the main features in SPH. The derivates only applies to the smoothing

kernel as the rest of the variables are constants. For example, the gradient only effect the smoothing kernel,

$$\begin{aligned} \nabla A_S(\mathbf{r}) &= \nabla \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \\ &= \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \ . \end{aligned} \tag{5}$$

The same property holds up for the laplacian,

$$\begin{aligned} \nabla^2 A_S(\mathbf{r}) &= \nabla^2 \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \\ &= \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \ . \end{aligned} \tag{6}$$

For a quick mathematic reference of gradient and laplacian see appendix B.

3.1.2 SPH fluid dynamics

In this section we will take a closer look at a model for simulating a viscous fluid. We will also examine what effect this model will have when used with the SPH method.

Fluid equations are often described in terms of force fields and density fields, but particles are usually considered as point masses and they reacts to point forces. Therefore we begin with examining the connection between point forces and force fields. We obtain the force acting on a particle i by integrating the force field acting on the particle,

$$\mathbf{f}_i = \int \mathbf{F} \ . \tag{7}$$

Approximating $\mathbf{F}$ with SPH interpolation over one particle with $\mathbf{F}_i = \mathbf{F}(\mathbf{r}_i)$ we get,

$$\begin{aligned} \mathbf{f}_i &= \int_{\mathbf{r}} m_i \frac{\mathbf{F}_i}{\rho_i} W(\mathbf{r} - \mathbf{r}_i, h) d\mathbf{r} \\ &= m_i \frac{\mathbf{F}_i}{\rho_i} \int_{\mathbf{r}} W(\mathbf{r} - \mathbf{r}_i, h) d\mathbf{r} \\ &= m_i \frac{\mathbf{F}_i}{\rho_i} \ . \end{aligned} \tag{8}$$

Refering to Newton's second law and by rewriting (8) we get the relations,

$$\frac{\mathbf{F}_i}{\rho_i} = \frac{\mathbf{f}_i}{m_i} = \frac{d\mathbf{v}_i}{dt} \ . \tag{9}$$

Usually the conservation of mass is described by,

$$\frac{d\rho}{dt} = -\rho \nabla \cdot \mathbf{v} \ , \tag{10}$$

referred to as the *continuity equation*. This is a trivial task in the SPH simulation. The particles in the simulation are explicitly forced to conserve the total mass in the system. In the simple case, the particles never changes their individual masses at all.

The Navier-Strokes equation formulates the conservation of momentum.

$$\frac{d\mathbf{v}}{dt} = -\frac{\nabla p}{\rho} + \frac{\mu \nabla^2 \mathbf{v}}{\rho} + \frac{\mathbf{F}}{\rho} \tag{11}$$

is a simplified version for a viscous incompressible fluid. $\mathbf{F}$ represents external force field and μ the viscosity of the fluid. There exists many forms of this equation in literature.

Dissecting the Navier-Strokes equation presented above, we can see two different internal components. The first component $(-\nabla p/\rho)$ represents the conservation of momentum, and the second $(\mu \nabla^2 \mathbf{v}/\rho)$ represents the viscous part of the equation. Observing the force field and point force relation in equation 9, and solving the SPH equations for the first internal component, we get the force acting on a particle due to momentum conservation, shown in the following equation,

$$\begin{aligned} \mathbf{f}_i^{\text{pressure}} &= m_i \frac{d\mathbf{v}_i^{\text{pressure}}}{dt} \\ &= -m_i \frac{\nabla p(\mathbf{r}_i)}{\rho_i} \qquad (12) \\ &= -\sum_j m_i m_j \frac{p_j}{\rho_i \rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \ . \qquad (13) \end{aligned}$$

This equation does not produce symmetrical forces between particles as, in general, $p_i \neq p_j$. Symmetrical forces are needed to get valid simulation results. Every reaction should have an equal sized counter reaction. Therefore we rewrite the momentum conservation component,

$$\frac{\nabla p}{\rho} = \nabla \left(\frac{p}{\rho} \right) + \frac{p}{\rho^2} \nabla \rho \ . \tag{14}$$

Replacing 14 in 12 we get symmetric forces,

$$\begin{aligned}
\mathbf{f}_i^{\text{pressure}} &= -m_i \frac{\nabla p(\mathbf{r}_i)}{\rho_i} \\
&= -m_i \left(\nabla \left(\frac{p(\mathbf{r}_i)}{\rho(\mathbf{r}_i)} \right) + \frac{p_i}{\rho_i^2} \nabla \rho(\mathbf{r}_i) \right) \\
&= -m_i \left(\sum_j \frac{m_j}{\rho_j} \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) + \frac{p_i}{\rho_i^2} \sum_j \frac{m_j}{\rho_j} \rho_j \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \right) \\
&= -m_i \left(\sum_j m_j \frac{p_j}{\rho_j^2} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) + \sum_j m_j \frac{p_i}{\rho_i^2} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \right) \\
&= -\sum_j m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \; . \qquad (15)
\end{aligned}$$

Equation (15) is clearly symmetrical. There exists other ways of balancing the forces to make them symmetrical. For example, [MCG03] suggests replacing p_j in equation (13) with $(0.5p_i + 0.5p_j)$.

To be able to calculate the forces for momentum conservation we need to know the pressure field, p. This is given by the *equation of state*, which often is used in its simplest form (see e.g. [DC96]), reminding of the ideal gas law ($p = k\rho$),

$$p = k(\rho - \rho_0) \; , \qquad (16)$$

where k is determined by the speed of sound in the material and ρ_0 is the rest density. Simulation of fully incompressible fluids often utilize more explicit methods for obtaining incompressibility, see e.g. [CR99].

Turning to the viscosity term and applying it to the the SPH equations we get,

$$\mathbf{f}_i^{\text{viscosity}} = m_i \frac{\mu \nabla^2 \mathbf{v}}{\rho_i} = \mu \sum_j m_i m_j \frac{\mathbf{v}_j}{\rho_i \rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \; .$$

Equation (17) suffers from a similar problem as equation (13). It does not produce symmetrical forces. In [MCG03] the following modification to (17) is suggested to balance the forces,

$$\mathbf{f}_i^{\text{viscosity}} = \mu \sum_j m_i m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_i \rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \; . \qquad (17)$$

Viscosity can be thought of as a measurement of how quickly fluid velocities converges to the mean velocity of the surrounding fluid. Viscosity forces are only depending on the differences in velocities, which makes equation (17) a natural way of force balancing.

Not included in the Navier-Strokes model are surface tension forces, which needs to be modeled separately. [MCG03] models these forces based on explicit ideas on ideas from [Mor00]. Surface tension struggles to minimize curvature by applying forces in the normal direction of the surface. Higher curvature creates greater forces.

First the *color field* is defined,

$$c_S(\mathbf{r}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \ . \tag{18}$$

Comparing (18) to equation (1) we can see that the color field interpolates the scalar value one at the sampled points. And as h is finite, the color field is zero outside the support radius from any particle ($\mathbf{X}_{outside} = \{\mathbf{x} : \forall \mathbf{r}_i, |\mathbf{x} - \mathbf{r}_i| > h\}$).

$$\mathbf{n} = \nabla c_S \tag{19}$$

defines a surface normal field, pointing into the fluid. And

$$\kappa = \frac{-\nabla^2 c_S}{|\mathbf{n}|} \tag{20}$$

defines the curvature of the surface. Putting it all together with a parameter, σ, for controlling the amount of surface tension we get,

$$\mathbf{f}_i^{\text{surface}} = \frac{m_i}{\rho_i} \sigma \kappa \mathbf{n} = -\frac{m_i}{\rho_i} \sigma \nabla^2 c_S \frac{\mathbf{n}}{|\mathbf{n}|} \ .$$

If we are going to construct a kernel function, we need to look back at the criterias for the smoothing kernels in equation 2 and 3. When construction a smoothing function, or selecting an existing, there are several properties which needs to be considered. This thesis aims at real-time implementation, therefore ease of calculation on the part of the CPU is an important factor. Another important factor are stability as a stable simulation enables larger time steps when integrating. Kernels should also have vanishing derivates

at the boundaries. With this in mind, [MCG03] suggest using the following smoothing kernels.

$$W_{\text{poly6}}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

have been chosen for ease of calculation. It is a simple formula and r only appears squared, so no square root needs to be calculated. Calculating the square root is an expansive operation on the CPU. For example, on recent CPUs, it is possible to invert one four by four matrix in less number of cycles then what is needed to calculate two square roots.

The size of the pressure force directly depends on kernel gradient length. W_{poly6} have vanishing gradient close to its center. Which is unwanted as if two particles gets to close together they will not have strength enough to repel each other. This causes clustering.

$$W_{\text{spiky}}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

has an increasing gradient length closer to its center. By using this kernel, clustering is avoided.

Viscosity can be thought of as smoothing the velocity field. Smoothing two velocities with each other causes their relative velocity to be reduced. Checking equation (17) it is seen that a negative laplacian increases the velocities between two particles. This is why

$$W_{\text{viscosity}}(\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 < r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

have been chosen, as it has a positive laplacian.

The derivates of these kernel functions can be found in appendix A. Only derivates of interest are presented.

3.2 Visualization

Presentation is important and it is hard to grasp a numerical representation. A fluid-like rendering of the simulated data will make the underlying simulation seem much more fluid like. This makes surface rendering an important part of this simulation. We could do volumetric rendering. But the

most defining visual feature of a fluid is its surface. This is why volumetric rendering is ignored. Surface rendering can be divided into two parts. Surface extraction and surface shading. Surface extraction extracts a surface approximation into a surface representation. The surface representation is then rendered and shaded. Shading defines the visual properties of a surface, for example properties like color, reflectance and refraction.

3.2.1 Surface extraction

Surfaces in this context are usually defined using a mathematical equality statement like $f(\mathbf{x}) = 0$. The surface is defined as all points that makes the equality statement true,

$$\mathbf{X}_{surface} = \{\mathbf{x} : f(\mathbf{x}) = 0\} \ . \tag{24}$$

This is called an iso surface or an implicit surface. A real world example in two dimensions is the coast line with altitude as the function. A negative altitude would indicate land below the water surface and positive altitude above the water surface. The coast line is where the altitude function equals zero.

The gradient of the function, at the surface, points in the direction of the surface normal.

One of the more classical methods for surface extraction is the marching cube algorithm. The marching cube algorithm was introduced in [WMW86], and developed by [LC87] for medical imagery. The marching cube algorithm generates triangles as surface representation. It discretizes space into cubes of uniform size. The function is evaluated at each corner of the cube. Each cube is classified by examining the sign of the corner values. Each corner can be negative or positive and there are eight corners which makes $2^8 = 256$ classification classes. Each class has a set of triangles associated with it. The associated triangles are used to approximate each cube. Triangle corners are adjusted by linear interpolation between the function values evaluated at the corners. The main disadvantage of using the marching cube algorithm in this context is that it introduces unnecessary connectivity information.

Another way of generating a surface, without the connectivity information, is to use ray casting to create three orthogonal layered depth images (LDIs) [SGHS98]. This is called the layered depth cube (LDC). This method was

introduced by Lischinski and Rappaport in [LR98]. Each LDI pixels store multiple points. These points corresponds to the ray-surface intersection points, where the ray is projected through the LDI pixel center.

In [dFdMGTV92] a physically based method for extracting an implicit surface is presented. There they scatter random particles within the search domain. Then they set up a differential equation which makes the particles migrate to the implicit surface when integrated. The differential equation involves applying forces on the particles in the direction of the function gradient. [dFdMGTV92] then creates a polygon model from the integrated particles. The final step of creating a polygon model is not necessary, as points will do just fine for rendering. Rendering surfaces from points will be discussed further on.

In [WH94], points are constrained to an implicit surface. They use repulsion, fission, fusion and adaptive size to evenly spread particles across a surface. They use large particles to quickly cover area that is later refined.

An improved scattering is presented in [DTC96] and [BW90]. If the function value is constructed from an underlying structure, a better scattering of initial particle positions can be made. Better here means shorter distance to surface. Also, purely numerical root finding algorithms can be used to migrate particles towards the surface instead of integrating a physical system.

3.2.2 Point splatting

Current graphical hardware uses mainly triangles to represent geometric structure and surfaces. This is why special care needs to be taken when rendering a surface which is represented by points on the surface. It is possible to generate triangles from the points by playing connect the dots. That would, however, waste precious processing power. A great survey has been written by Jaroslav Křivánek on the topic of representing and rendering surfaces with points [Jar03].

The points on the surface, that form the surface representation, are usually referred to as surfels (surface elements). To cover the surface, all surfels are assigned a small area. Common properties held by each surfel are:

In the good old days, before powerful programmable GPUs, transforming and blending could be heavy duty tasks. This is why opaque quads have been used as rendering primitive in early point rendering pipelines. Quad side

position	We would not be able to do much without this.
normal	Used for shading and for projection to screen space.
radius	Used when the surfel represents a circular area in object space.
color	Most materials have color variations across the surface.

length would be given by $2rs$, where s is a scaling factor given by perspective projection and viewport transformation and r is the radius of the disc.

Next evolutionary step of the point as primitive rendering pipeline is to use opaque ellipses as rendering primitive. Ellipse parameters comes from projecting an object space disc to screen space. It is comparable to flat shading when using a triangle mesh for rendering.

Previous two mentioned rendering primitives do not do any interpolation between the surfels. Interpolation is crucial for high quality rendering. This leads us to splatting. Splatting uses an alpha mask (often a truncated 2D Gaussian function) and has additive blending. The alpha mask (weight function) is defined in object space and then projected into screen space. Resulting formula is

$$A(x,y) = \sum_i A_i w_i(x,y) \ , \tag{25}$$

where A_i is a property of splat i which needs splatting. And w_i is the projected weight function of splat i at screen position (x,y). A_i is often color. But, when using more advanced shading methods, other properties like surfel normal can be used for interpolation. The above formula is most often not suited for direct use, as, in general, all the weight functions does not sum up to one:

$$\sum_i w_i(x,y) \neq 1 \ . \tag{26}$$

This results in an uneven interpolation. Of course, we don't want an uneven interpolation, and that is why normalization is needed. The adjusted normalized formula is

$$A(x,y) = \frac{\sum_i A_i w_i(x,y)}{\sum_i w_i(x,y)} \ . \tag{27}$$

On legacy hardware, per pixel normalization is not supported. That is why there exists per surfel normalization techniques. Luckily the author of this thesis did not have to deal with old hardware, and these methods will not be presented.

Pseudo code for splatting can be seen in figure 1. Shading is done per surfel and the color result of the shading is interpolated.

```
Splatting() {
  for each surfel {
    project surfel to screen space
    shade surfel
    splat shade output color
  }
  for each pixel {
    normalize pixel color
  }
}
```

Figure 1: Splatting using object space shading.

A major drawback of using per surfel shading is that in order to get high frequency output a really dense sampled surface is needed. This is similar to using Gourad shading when comparing to triangularized surface representation. Too many surfels lead to low frame rates. To remedy this problem it can be wise to do screen space shading as shown in figure 2.

```
Splatting() {
  for each surfel {
    project surfel to screen space
    splat surfel attributes
  }
  for each pixel {
    normalize pixel attributes
    shade pixel using pixel attributes
  }
}
```

Figure 2: Splatting using screen space shading.

Depending on shader, there are a number of different properties that might be interpolated. If using an advanced shader several properties might be needed. As an example of a simple shader we look at rendering a completely reflective material. The reflected vector is calculated at each surfel and then interpolated. All that needs to be done, per pixel, is to use the interpolated reflected vector as index into an environment map.

If we have a densely sampled surface resulting in surfels being small compared to pixel size we get aliasing effects. Nyquist's theorem says: In order to avoid

aliasing effects, sampling rate must be twice as high as the maximum frequency in the signal. It is obvious that it is impractical to increase sampling rate (we can not have infinite screen resolution). In ordinary signal processing the signal is usually filtered with a bandwidth limiting filter before sampling. This is why EWA (Elliptical weighted average, [GH86]) was proposed, by Zwicker in [ZPBG01], to be used in this context. EWA point-splatting basically works by projecting surfels to screen space and then do a mathematical screen space convolution on the projected screenspace weight function with a smoothing function. The convoluted screenspace weight function is then used for splatting. The mathematical convolution is analogous to an analog bandwidth limiting filtering. Convolving two Gaussian functions results in a Gaussian function. This makes them relative easy to handle and that is why Gaussian functions are often used both as convolution functions and as weight functions, respectively.

3.2.3 Shading

Shading defines material properties. The schema of a simple light model for translucent materials can be found in figure 3.

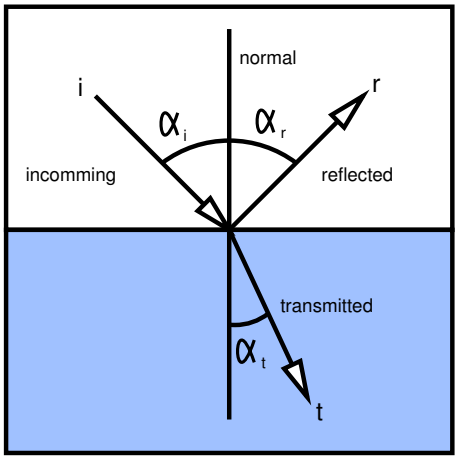

Figure 3: Schema of a simple light model.

The following relations hold for this model,

$$cos(\alpha_i) = cos(\alpha_r) = -\frac{\mathbf{i} \cdot \mathbf{n}}{\|\mathbf{i}\|\|\mathbf{n}\|} = \frac{\mathbf{r} \cdot \mathbf{n}}{\|\mathbf{r}\|\|\mathbf{n}\|} \ . \tag{28}$$

These relations means that the light is reflected in the same angle as the

incoming light. Expanding this to a vector equation we get,

$$\mathbf{r} = \mathbf{i} - 2\frac{\mathbf{i} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}\mathbf{n} \, . \tag{29}$$

For the ray being transmitted into the new media, the amount of refraction is described by Snell's law,

$$n_i \sin(\alpha_i) = n_t \sin(\alpha_t) \, , \tag{30}$$

where n_i and n_t are indices of refraction (IOR). IOR are measured values and varies depending on a lot of factors such as light wavelength and temperature. IOR for air is about one and the IOR for water can be seen in table 1. As seen in the water IOR table the indices varies just slightly and this variation is usually ignored.

wavelength	700 nm (red)	530 nm (green)	460 nm (blue)
IOR at 20°C	1.33109	1.33605	1.33957

Table 1: The IOR of sweet water at 20°C.

By observing water we can see that the amount of reflection depends on viewing angle. The probability that a photon is reflected, R, can be calculated with

$$R = \left(\frac{sin(\alpha_i - \alpha_t)}{sin(\alpha_i + \alpha_t)}\right)^2 + \left(\frac{tan(\alpha_i - \alpha_t)}{tan(\alpha_i + \alpha_t)}\right)^2 \, . \tag{31}$$

It is a result of research by Augustin-Jean Fresnel and it is called the Fresnel equation [Fre]. The equation shown here refers to non-polarized light.

A common approximation to the Fresnel term is

$$R_{approx} = (1 - \mathbf{i} \cdot \mathbf{n})^n \, , \tag{32}$$

where both $\mathbf{i}$ and $\mathbf{n}$ are normalized ($\|\mathbf{i}\| = \|\mathbf{n}\| = 1$). When comparing R and R_{approx} as a function of α_i, it is seen that this is a reasonable approximation.

3.3 Algorithmic issues

Main computational algorithmic issue is the proximity query needed to determine interacting particles. Algorithms that was taken into consideration are presented below.

3.3.1 Spatial query algorithms and data structures

During simulation, all interacting pairs needs to be determined. The naive solution would be to loop over all pairs of particles to determine which of those are sufficiently close for interaction. Execution time would be $O(n^2)$ in an environment of n elements and would produce a bottleneck. This is a classical problem in collision detection. The field of collision detection are far to comprehensive to be covered here, and mostly irrelevant for this problem (Those interested can start with [Eri04]). Presented below are the methods considered for generating the sphere-sphere intersection pairs needed for the simulation. The presented methods do not generate the exact sphere intersection directly and are usually referred to as the *broad phase*. The *narrow phase* performs the exact detection on the pairs generated by the broad phase. In this case the narrow phase will consist of sphere-sphere intersection tests which is considered trivial and is omitted. Whenever possible, the broad and narrow phase is merged as optimization.

Concerning the broad phase, two classes of methods exist. Direct spatial addressing, which gives directly accessible structures, and spatial subdivision, which results in a tree data structure that can be traversed relatively fast.

The simplest form of spatial data structure would be to discretesize the position of each particle to fit it into a three dimensional matrix. This is done by scaling and offsetting the position and then rounding to the nearest integer. These integer coordinates are then used to index into the matrix. Clipping needs to be done if the integer coordinates exceed the dimensions of the matrix. The particle is then added to the corresponding position in the matrix. Each element in the matrix now contains all particles found in the corresponding piece of the grid. When finding the interacting partners for a particle, only neighboring cells and the cell containing the particle needs to be searched. If m is the medium number of particles per cell then the time complexity would be reduced to $O(nm)$.

Instead of addressing into a matrix, a hash table can be used. Hashing on the integer coordinates instead of addressing into the matrix result in a really large virtual matrix. This also makes it more effective to to traverse cells of interest, as only cells with particles in them are kept in the hash table. This method is presented in [THM+03].

We now take a closer look on sorted structures, starting with the octree. Octree is among the more commonly used spacial structures. It works by

subdividing axis aligned cubes. The top level consists of a single cube which surrounds all particles. Each cube is then recursively defined to contain either eight sub cubes (hence the name octree) or the set of particles inside the cube. There are two main criterias used to decide when to stop recursing. It can be decided on the basis of how many particles are contained in the cube or on the number of recursions. The construction of an octree generally takes $O(n \log n)$ time units. Using spacial frame coherency this can be reduced to $O(n)$ for updating the the tree and generating the pairs of possible collision. This has been done in [BYL98].

Another tree structure that is commonly suggested in similar contexts is the k-d tree [Ben75]. The k-d tree is a balanced binary search tree that uses k number of keys. In this application the coordinate axis are used as keys. When constructing the tree, an axis is selected for each internal node. There are two main approaches to select a key for each node (in this case the x, the y or the z axis). The most simplistic way is to use alternating keys depending on the level of the node. So at level L, key number $L \mod k + 1$ is used, root being level 0. The root node represents the entire box volume spanned by the particles. The volume represented by each child node is the result of splitting the parent volume along the plane formed by the parent key axis and parent position. To keep the particles in each sub volume as close together as possible, it is often suggested to use the key axis which corresponds to splitting the volumes along the longest side of the box. There is no need for storing child pointers in a k-d tree. Nodes are preferably stored in a simple vector and if the root node have index 1 then each node with index n can find its children on index $2n$ and $2n + 1$. This is similar to how a heap is usually implemented. Constructing the tree is usually a $O(n \log n)$ operation. Main features of a k-d tree are the compact representation, its simplicity and that it is balanced.

The sweep and prune algorithm presented in [CLMP95] keeps track of all overlapping *axis aligned bounding boxes*. The algorithm works by keeping three sorted lists (one for each coordinate axis). The values in each list corresponds to the minimum and the maximum of each bounding box projected onto corresponding axis (see figure 4). Each list now contains information about which intervals overlap in each dimension. A pair of bounding boxes intersect *if and only if* their intervals overlap in all three lists. In general sorting takes $O(n \log n)$ time units. However, when dealing with a nearly sorted list, a complexity approaching $O(n)$ is achieved using bidirectional bubble sort, also known as shaker sort. When looking at the frame to frame coherency, one can see that small changes to the bounding boxes cause small

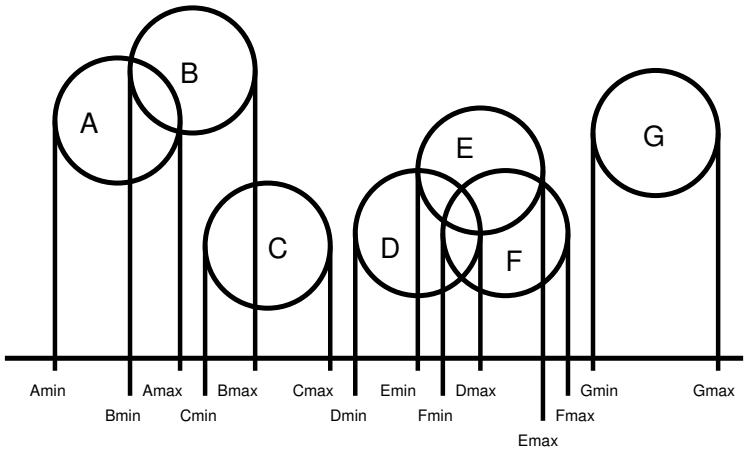

Figure 4: Projected min and max values

changes in the sorted lists. The sweep and prune algorithm relies on this coherency for its efficiency. In addition to the sorted lists, the algorithm keeps two sets of bounding volume pairs. One set which holds all fully intersecting pairs, and one which holds all potentially intersecting pairs, which means at least all pairs that overlap one or more dimensions. The set of potentially intersecting pairs also needs to keep track of overlapping status. In [CLMP95] it is suggested to use three flags, one for each dimension, with overlapping status, although it would be sufficient to count the number of dimensions in which overlapping occurs. Whenever all three dimensions overlap the pair is also found in the set with the fully intersecting pairs. The overlapping status is updated while resorting the lists. Whenever a swap between a minimum and a maximum value occurs in the sorting algorithm it corresponds to a change in overlapping status for that axis. The change is updated in the potentially intersecting pairs set and if needed in the fully intersecting pairs set.

The validity of the sweep and prune algorithm in this context can be considered a little uncertain. It has good support for dynamic size of the underlying smoothing kernel and the number of pairs going to the the narrow phase are kept fewer then at least the grid based methods. However, Its efficiency depends heavily on frame to frame coherency and could in worst case become $O(n^2)$. Although the $O(n^2)$ could perhaps be limited by limiting the number of iterations in the resorting phase and reinitializing the lists and sets with other methods if fully sorted lists were not obtained within the limit. However, that would need further analysis. Therefore, testing in real situations is needed to determine the coherency.

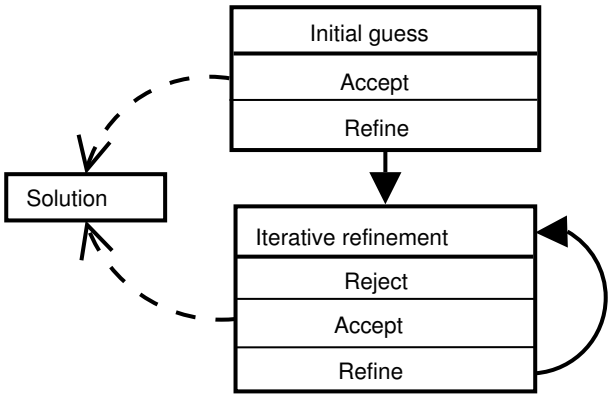

Figure 5: General algorithm for generating the surface approximation.

4 Implementation

4.1 Smoothed Particle Hydrodynamics

Implementation is mainly based on [MCG03]. Main difference in SPH simulation is in the balancing of pressure forces. This implementation uses equation 15 to get symmetrical forces.

4.2 Visualization

4.2.1 Surface extraction

We will approximate an iso surface with overlapping circular discs (surfels) for rendering (surface splatting). The scalar field, from which the iso surface is extracted, is produced by local functions around points in a point cloud. The scalar field is defined to be zero everywhere else. This means that the surface only exists near the points which produce the scalar field. We will use the color field from the SPH simulation as the scalar field (equation 18). Taking into account that the surface is intended for rendering, visual culling is added to the algorithm for optimization. A general outline of the algorithm can be seen in figure 5. First a general algorithm will be described and then current implementation will be presented in detail. Also, an example in two dimensions is presented.

All the operations are working on sets of surfels. When solution and initial

guess is mentioned it is in terms of sets of surfels, unless otherwise stated.

As this is an iterative solution, first step is to generate an initial guess of the surface. A good guess is important in order to find a good solution. If the initial guess does not cover the solution surface, then the iterative refinement process will probably not cover the surface either. If the initial guess covers too much area, then CPU time will be wasted. The initial guess only needs to cover the potentially visually important areas of the surface. In some cases, parts of the initial guess can be considered good enough. For example, when only one particle can be found in the local neighborhood, the iso surface approximation can be precalculated for that function and added to the solution without need for further refinement.

The iterative refinement process works by calculating the function and gradient of the scalar field at the center of each surfel. Then each surfel is adjusted with one Newton-Raphson step,

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{f(\mathbf{x}_n)\nabla f(\mathbf{x}_n)}{|\nabla f(\mathbf{x}_n)|^2} \ , \tag{33}$$

in the direction of the gradient, in order to get a closer approximation to the surface. The function f is defined from a constant, $k_{isocolor}$ (will be discussed further down), and the colorfield,

$$f(\mathbf{x}) = c_S(\mathbf{x}) - k_{isocolor} \ . \tag{34}$$

The length of the step for each surfel is an approximation of the error in placement for that particular surfel. If the approximated error is considered too great, then the original placement can be considered faulty and the surfel is altogether discarded from the solution.

After faulty surfels have been discarded, the remaining set of surfels are divided into two sets. One in which all surfels are considered a good enough approximation for their part of the surface, and the other one with surfels that needs further refinement. The set of good enough surfels are added to the solution. In the other set, all surfels are divided into smaller surfels (figure 6). This set is then used as a refined guess. The refinement process is iterated until the set of surfels needing refinement is empty, or until a predefined number of iterations has been reached. When the iteration stops, all remaining surfels are added to the solution. Surfel splitting does not occur in the last iteration.

Here follows the description of current implementation for extracting an iso surface from the color field of an SPH simulation. We want to visualize a

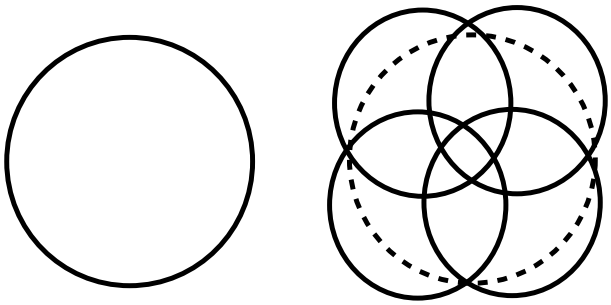

Figure 6: Surfel needing refinement are split into smaller surfels.

surface that surrounds the simulated particles. By observing that the color field (equation 18) interpolates the scalar value one, we know that the color value at each simulated particle to be approximately one. We also know the color field to be zero everywhere else but close to simulated particles. An iso value of 0.5 is chosen, and as the color field function is smooth we get a surface that surrounds the simulated particles. Visual inspection of simulation confirms this to be a decent value. From this, it can be deduced that the surface are defined on an approximate distance from the surface SPH particles of half the length of the kernel function used.

First we will find all surface particles and then add a circular surfel to the initial guess set for each surface particle. The surfel should be placed at a distance of half the length of the kernel function from the originating particle. The direction of the placement is in the direction of the steepest descent of the color field (the gradient direction). The color value and color gradient of each particle are already calculated for each simulated particle. Those values are used to calculate one Newton-Raphson step from each particle. The step length is an approximation to how close the surface lies to that particle. If the step length exceeds a threshold, the particle is considered to be an internal particle and is ignored. If the Newton-Raphson step is closer to the particle then half the kernel radius that point will be considered to be a better guess for a point on the surface than a point with a distance of half the kernel radius from the originating particle. For detail control, the laplacian is examined as a measure of the curvature on the surface. If the laplacian value exceeds a threshold (e.g. a curvy surface) the surfel is split and added to the initial guess as in figure 6. Only front facing surfels are added to the initial guess for better utilization of the CPU.

This will however not cover certain cases like single particles or when particles form a plane or a line. It will not cover cases when the surface can be found

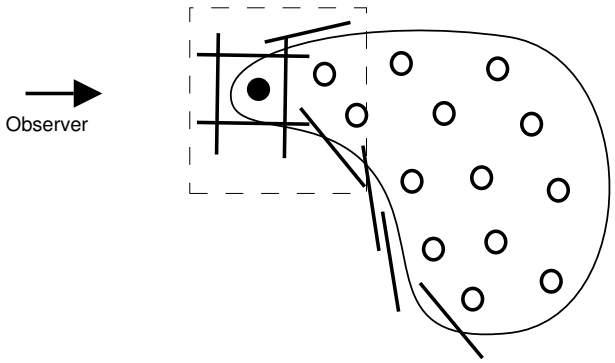

Figure 7: How an initial guess could look like in 2D.

in more than one direction around the particle. To handle this situation the local covariance matrix is calculated around each particle,

$$\mathbf{C} = \sum_{i=1}^{n} (\mathbf{x}_i - \mathbf{x})(\mathbf{x}_i - \mathbf{x})^T \ , \tag{35}$$

and its properties are examined similar to [MKN+04]. If the covariance matrix has one or more eigenvalues close to zero then the particle is surrounded by surface on at least on two sides. To determine if one of the eigenvalues is close to zero the determinant of the covariance matrix is calculated. The determinant is the same as the product of the eigenvalues, and if one of the eigenvalues are close to zero the product of them will be close to zero too. To normalize the value of the determinant, the mean value of the eigenvalues is calculated by taking the trace of the covariance matrix. The trace is the same as the sum of the eigenvalues, so the mean value of the eigenvalues is the trace divided by three. Normalization is done by taking the determinant divided by the mean value to the power of three,

$$P_{normalized} = \frac{det(\mathbf{C})}{(trace(\mathbf{C})/3)^3} \ . \tag{36}$$

If this value is close to zero, then then surface can exist on any side of the particle. In that case surfels are generated to surround the particle. Front facing surfels are added to the initial guess. Generation of both six (sides of a box) and eight (corners of a box) surfels for covering the potential surface around the particle have been tested.

Figure 7 illustrates how an initial guess could look like in two dimensions. The circles represent particles and the thin line represents the iso surface (iso

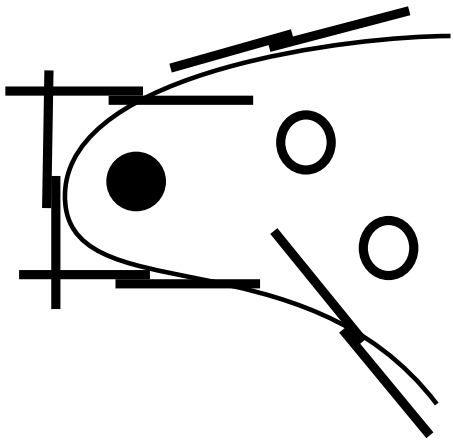

Figure 8: How the working set could look like in 2D.

line in 2D). The filled circle represent a particle which has been classified to generate surface all around it. As can be seen, it generates one surfel inside the volume (area) surrounded by the surface (line). That surfel would in this case normally be back face culled before added to the initial guess, but it is kept it for illustrational purposes. It will be be removed further on in the refinement step. Back face culling can be seen by the lack of surfels facing away from the observer (not including the above exception). The working set does not keep the direction of each surfel, it only keeps track of size. The orientations of the surfels indicated in the figure are defined as being perpendicular to the surfels latest offset direction. The same orientation is used when adding the surfels to the solution and to define the tangent plane used when splitting the surfels.

In the iterative refinement step all bad guesses are first discarded by checking the Newton-Raphson step length. Surfels are split and the algorithm is iterated a constant number of iterations (about one or two iterations). A good criteria for sorting out surfels to add to the solution before the specified iteration level is reached have not been found. Some detail control have been implemented in the first step when generating the initial guess.

An illustration in two dimensions on how the refinement process works can be seen in figures 8 and 9. They show the area marked by a dashed rectangle in figure 7. Figure 8 shows the working set with one Newton-Raphson step taken and splitting have taken place. The misplaced surfel has been removed in the Newton-Raphson step. The divided surfels are misaligned in the figure for clarity. Normally they would form a straight line.

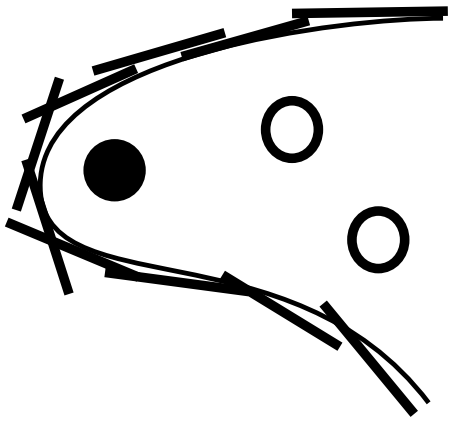

Figure 9: How the solution approximation could look like in 2D.

The result shown in figure 8 is fed into the final iterative refinement step. No subdivision is done after the Newton-Raphson step. The result is added to the solution shown in figure 9.

There are problems with covering the entire visible surface in the initial guess. Because of this, small holes can be noticed in the surface representation. This is due to either a bad guess or completely missing a part of the surface. As the surface is regenerated each frame and when simulating fluids the surface is far from static the holes will never be long lived. There is no checking of the distribution of generated surfels and local parts of the surface with excessive number of surfels can occur.

For future improvements the first thing to improve would be the initial guess. It might be worth trying to use only the local distribution to generate a good guess. When the simulated particles form a plane then the implemented algorithm generates six or eight surfels for each particle (not considering back face culling). In a plane only two would be needed. This is a waste of computation and could be improved. Another way of generating the initial guess could be to use some coarse regular method, or a stochastic method, for finding a surface approximation and feed that into the refinement process. This could solve problems with covering the entire surface. To find a criteria for sorting out surfels needing further refinement in the iteration process is a good candidate for further investigation also.

We have also implemented an algorithm for generating a LDC type of representation. This was implemented as a reference algorithm for comparison. As ray-casting is an expensive operation we use a ray-casting approxima-

tion with similarities to the marching cube algorithm. In every position in a regular grid, near the particles, the scalar function is evaluated. The positions in the regular grid correspond to the intersection of three rays, one ray from each of the three orthogonal LDIs. We then traverse all rays via the grid points they intersect, but only evaluated grid points are traversed. If the next grid point to be traversed has different sign then current, we know there have to be at least one intersection between those two grid points. One intersection is assumed and the position of this intersection is determined by interpolating the values at the grid points involved. We also interpolate the gradient evaluated at the grid points to get a normal direction at the intersection. For optimization, we have changed the traversal order to traverse all grid points only once. For each grid point, the next grid point in each direction is examined before continuing.

4.2.2 Surface rendering

The surface extraction algorithm created generates points as output. This is why point splatting was selected as rendering method. The alpha mask is,

$$w_{\text{pointsplat}}(r,h) = \frac{1}{h^6}\begin{cases} (h^2-r^2)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}, \tag{37}$$

where h is determined by each surfel and r is the radius from surfel center.

Image based shading is used and the only property splatted is the reflection vector. The Fresnel term is approximated by

$$R_{approx} = (0.5 + 0.5\mathbf{r}\cdot\mathbf{l})^2 , \tag{38}$$

where $\mathbf{r}$ is the normalized reflection vector and $\mathbf{l}$ is the normalized eye vector.

It is definitely not the best approximation, but it does the trick of smoothly interpolating between non-reflective, when looking straight down into the water, and fully reflective, when looking at the water at a right angle. It was chosen over a more accurate equation to simplify the per pixel calculations.

For refraction a non physical model is used. We want to have a distorting effect depending on surface normal. To achieve this the background behind the fluid is sampled as follows,

$$c_{refracted}(x,y) = t_{back}(x - \mu r_x, y - \mu r_y) , \tag{39}$$

where r_x and r_y are the x and y components of the reflection vector. μ is a small constant to control the amount of distortion. To fully use the light model presented in 3.2.3 would require ray-tracing to render all internal reflection and refraction effects.

The resulting color of the fluid surface can be calculated from,

$$c_{out}(x,y) = (0.5 + 0.5\mathbf{r}\cdot\mathbf{l})^2 t_{env}(\mathbf{r}) + c_{fluid} t_{back}(x - \mu r_x, y - \mu r_y) \ , \quad (40)$$

where t_{env} is an environment map function, c_{fluid} is an approximation for the light absorbed by the fluid and t_{back} is the background texture function. It could be done in a more physically accurate way, but this has not been the main focus of this thesis.

4.3 Algorithmic issues

For neighbor detection, uniform spatial hashing was used. It worked satisfactorily and no other algorithm was implemented.

A general optimization strategy used is to batch-process as much as possible. This reduces function calling overhead and prepares for a parallel implementation.

4.3.1 System design

The implementation of this system is divided into five parts, four libraries and one executable. This is done to keep different parts of this implementation as separate as possible. The reasons for keeping separate parts are to keep maintenance and simple code simple.

The first library is called *mzbase* and contains classes that can be used in any project. Example of these classes are reference counting base classes, singletons and extendable objects.

The second library contain some simple mathematics, *mzalgebra*. Right now it mostly contains simple vector and matrix classes to keep track of positions, directions and rotations.

The third library is one, out of two main libraries, called *mzparticles*. This is where all things concerning particle simulation is kept. Here is where you find

things like classes for calculating particle forces and for integrating particle positions.

The fourth library is the second main library, *osgwrapper*. This where the rendering takes place. This is also where you find the surface extraction. Here we have the first external dependency, as this library uses OpenSceneGraph for rendering [Ope]. Mostly opengl functions are called directly in this library. This library only presents geometry to the GPU and shading is left to the application.

And finally we have the application. This is where all thing are tied together. An example simulation is set up with initial parameters. This is also where rendering and shading is controlled.

In order to get a flexible and effective simulation, the design of the simulation loop was modeled according the principles of a conveyor-belt. For example, in a car factory, the chassis of one car starts at one end of the conveyor-belt, stopping at each station along the belt for modifications. Modifications like mounting a motor or painting the car. In the case of our particles, a particle container is considered to be our chassis. For each time step in the simulation loop the particle container is fed into the "conveyor-belt", or *ProcessorProgram* as it is referred to in the code. Each *ProcessorProgram* contains *Processor*s and each *Processor* modifies the particle container in some way. Example of what *Processor*s can do are adding and removing particles to/from the simulation, or reseting forces or calculating pressure on the particles or integrating. The particle container have been designed to be extensible, so *Processor*s can add additional information the particle container for other *Processor*s to use. *ProcessorProgram*s are also *Processor*s so *ProcessorProgram*s can be recursively defined. This makes this design very flexible. This design is also effective as the particles are batch processed for each *Processor*. This means less function calling overhead and some variables only needs to be set up once for each batch.

5 Results

5.1 Smoothed Particle Hydrodynamics

Simulating 2200 particles on a Pentium 4 1.6GHz (mobile, not the m model) takes 22ms per frame. This is similar to the results of the CPU version

in [AIY+03]. The difference is they are using an Intel Pentium 4 2.8GHz. This thesis has virtually no focus on implementation optimization, only algorithmic optimization. We believe we can get similar simulation speeds, or better, on the CPU as [AIY+03] has with their GPU version. This would be done by optimizing using the SSE feature which exists on recent CPUs. SSE instructions are SIMD[2] instructions.

5.2 Visualization

5.2.1 Surface extraction

Table 2 shows the timings of the surface approximation generation. The timings where taken from a simulation of 2200 particles on a Pentium 4 mobile (not same as the Pentium 4 m) 1.6GHz, rendered without pixel and vertex shaders. An LDC method for generating the surface was implemented as reference. It shows surface generation timing, number of surfels generated, number of function evaluations needed and the frame rate for the entire simulation. During a simulation these values vary with the total surface area as a result of the dynamics of the system, but typical values have been selected.

method	generation	surfels	evaluations	full frame
1 iteration	17 ms	900	1100	17-22 fps
2 iterations	40 ms	4000	5500	10-14 fps
3 iterations	100 ms	19000	25000	5-7 fps
LDC	130 ms	4000	10000	5-7 fps

Table 2: Surface generation timings

When reading table 2 keep in mind that the iterative method generates mostly front facing surfels. Comparing three iterations with the LDC method shows a huge difference. Nineteen thousand mostly front facing surfels compared to four thousand front and back facing surfels at the same frame rate. That is a huge difference. One might think this depends mostly on code optimizations, but by ignoring frame rate and assuming algorithm efficiency depends on generated surfels compared to number of function evaluations of the scalar field the degree this is not an issue. And when comparing efficiency in this way an efficiency of 70-80 percent is reached using the iterative

[2]SIMD stands for Single Instruction Multiple Data

method compared to 40 percent using the LDC method. And that is when *ignoring* the difference that LDC generates both back and front facing surfels. The LDC algorithm can be algorithmically optimized by adopting a method similar to the marching cube surface searching found at [And01]. This would bring the efficiency up to about 50 percent independent of surface resolution. Increasing surface resolution in the LDC algorithm without using the surface search would decrease the efficiency. Considering the front and back facing issue would imply the iterative method to be about 200 percent *more* efficient. When generating high quality surfaces, the efficiency of the iterative algorithm approaches 75 percent using previous definition of efficiency.

In addition to being more efficient, the iterative method produces more accurate results in this application. A surface generated from two iterations was compared to LDC generated surface of similar surface resolution. The medium approximated uncertainty of the surfel centers differed by a factor of three to the disadvantage of the LDC algorithm.

5.2.2 Surface rendering

Visually plausible results has been achieved. A screenshot from the test-application build can be seen in figure 10. More media from this application can be found at [Ves04]. Current and future developments will be found at [VRL].

There is room for improvement concerning rendering speeds. Rendering 19000 surfels in 30ms is far from great. Although this should be compared to 100ms for generating the 19000 surfels. For each vertex, several gl functions are called for uploading vertex properties onto the GPU. By using vertex buffers vertex property uploading times could be reduced. Although unless surface generation is moved to the GPU, uploading vertex properties will still present a bottle-neck.

6 Conclusions and future work

Nearing the end of this thesis it is now time to sum up and look forward. We have seen the implementation of a system for simulation and rendering a viscous fluid using smoothed particle hydrodynamics in real time. Using very coarse material approximation we still get plausible results. A new up-

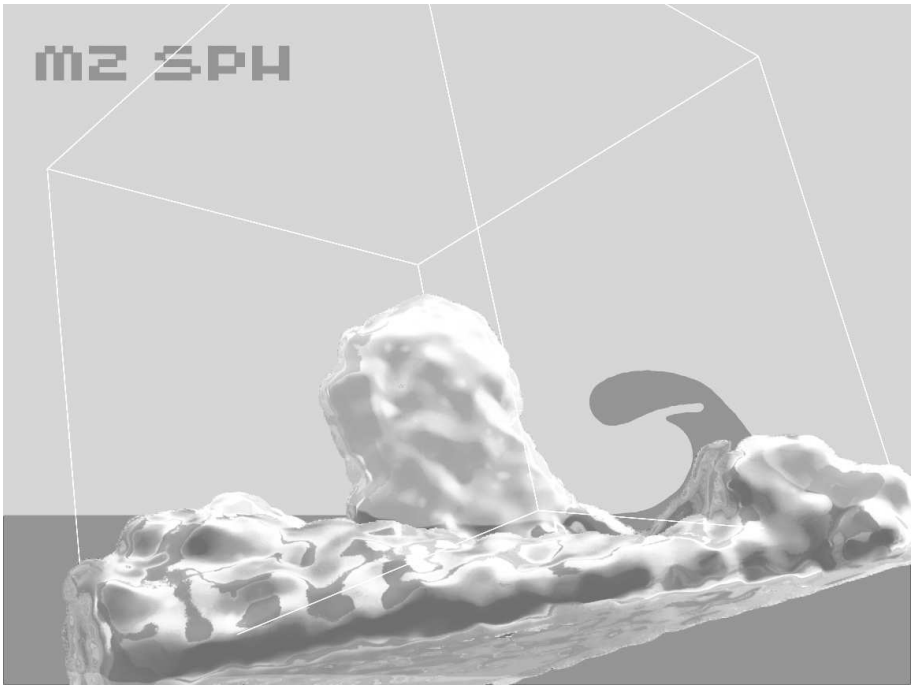

Figure 10: A screenshot taken from test application.

sampling technique has been presented. Comparing to other similar software implementations, our solution seems a little bit more optimized, although it is a bit hard to tell.

Considering the high flexibility of todays GPUs there are no excuses for using as simple material approximation as we do. This is an easy fix in future implementations. The main intent of using a more simple material approximation where to enable implementation in legacy hardware, more specifically first generation programmable GPUs. This might be worthy a quick investigation.

Our new surface refinement technique is trivially parallel, and very suited for GPU implementation. We feel it is more suited for GPU implementation then implementing the simulation on the GPU. In [AIY+03] collision detection is calculated on the CPU. This mean reading simulation results from GPU memory and, even thou they get good results, it is usually unwanted to read from the GPU memory. When doing surface refinement on the GPU there is no need of reading back the result to main memory as the result is also presented by the GPU. Also, when comparing the timings of simulation and generating the surface one can see that generating the surface is in greater

need of optimization. GPU implementation of surface generation would be very interesting to see.

CPU optimization techniques should be examined for increased performance. By using ordinary CPU optimization techniques such as using SIMD instructions, and perhaps some loop unrolling, we believe we could at least cut the simulation time in half. Cache efficiency does not present a problem in our simulation as only a few thousands of particles are used, but when dealing with upsampled surface is another issue. So, CPU and cache optimizations would definitely increase performance.

This implementation is now used as basis for further research at VRlab, Umeå University. Their main objective is to simulate sand and earth moving in vehicle simulators. SPH is considered to be an outstanding method for these applications. The goal of the thesis project has been to implement a framework that allows for additional research and development. As a next step, models for elasticity, plasticity and adaptive simulation resolution will be implemented. Other potential application areas include e.g. soft tissue simulation for haptic and medical simulators as well as special effects in computer games and movies. For more details see [VRL].

7 Acknowledgments

First, I would like to thank Kenneth Holmlund his feedback and help during the work on this master thesis. I would also like to thank Anders Backman for helping with the OpenSceneGraph api [Ope].

References

[AIY+03] Takashi Amada, Masataka Imura, Yoshihiro Yasumuro, Yoshitsugu Manabe, and Kunihiro Chihara. Particle-based fluid simulation on gpu, 2003. http://chihara.naist.jp/people/2003/takasi-a/research/index.html visited 2004/10/15.

[And01] Andreas Jönsson. Fast metaballs, April 2001. http://www.angelcode.com/dev/metaballs/metaballs.asp visited 2004/09/02.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BW90] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 109–116. ACM Press, 1990. http://www.unchainedgeometry.com/jbloom/pdf/interactive.pdf visited 2004/09/21.

[BYL98] Vemuri B.C., Cao Y., and Chen L. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17(2):121–134(14), June 1998.

[CJ91] James Cameron and William Wisher Jr. Terminator 2: Judgment day, 1991. http://www.imdb.com/title/tt0103064/ visited 2004/10/17.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995. http://haptic.mech.nwu.edu/library/cohenj/acm95/ visited 2004/09/03.

[CR99] Sharen J. Cummins and Murray Rudman. An sph projection method. *J. Comput. Phys.*, 152(2):584–607, 1999.

[DC96] M. Desbrun and M. P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In

Computer Animation and Simulation ´96 (Proceeding of EG Workshop on Animation and Simulation), pages 61–76. Springer-Verlag, aug 1996.

[dFdMGTV92] Luiz Henrique de Figueiredo, Jonas de Miranda Gomes, Demetri Terzopoulos, and Luiz Velho. Physically-based methods for polygonization of implicit surfaces. In *Proceedings of the conference on Graphics interface '92*, pages 250–257. Morgan Kaufmann Publishers Inc., 1992. `http://www.visgraf.impa.br/Projects/dtexture/pdf/phys.pdf` visited 2004/09/21.

[DTC96] Mathieu Desbrun, Nicolas Tsingos, and Marie-Paule Cani. Adaptive sampling of implicit surfaces for interactive modeling and animation. *Computer Graphics Forum*, 15(5), dec 1996. Published under the name Marie-Paule Gascuel. `http://www-grail.usc.edu/pubs/DTG_CGF96.pdf` visited 2004/09/21.

[Eri04] Christer Ericsson. *Real-Time Collision Detection*. Morgan-Kauffmam, 2004.

[Fre] Fresnel equations. `http://scienceworld.wolfram.com/physics/FresnelEquations.html` visited 2004/12/02.

[GH86] N. Greene and P. Heckbert. Creating raster omnimax images from multiple perspective views using the ellipticalweighted average filter. *IEEE Computer Graphics & Applications*, 6(6):21–27, June 1986.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics. *Monthly Notices of the Royal Astronomical Society*, (181):375–389, 1977.

[GPG] Gpgpu - general-purpose computation using graphics hardware. `http://www.gpgpu.org/` visited 2004/10/17.

[Gro01] M. Gross. Are points the better graphics primitives? *Computer Graphics Forum*, 20(3), 2001. Plenary Talk Eurographics 2001.

[Jar03] Jaroslav Křivánek. Representing and rendering surfaces with points, February 2003. `http://www.cgg.cvut.cz/~xkrivanj/` visited 2004/10/15.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[LL03] G. R. Liu and M. B. Liu. *Smoothed Particle Hydrodynamics ? a meshfree particle method*. World Scientific Publishing, 2003.

[LR98] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Rendering Techniques '98*, pages 301–314, June 1998.

[Luc77] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astrophysical Journal*, (82):1013–1024, 1977.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159. Eurographics Association, 2003.

[MKN+04] Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point based animation of elastic, plastic and melting objects, 2004. `http://www.pointbasedanimation.org/` visited 2004/08/24.

[Mor00] J.P. Morris. Simulating surface tension with smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 33(3):333–353, 2000.

[Ope] Openscenegraph - the high peformance open source graphics toolkit. `http://www.openscenegraph.org/` visited 2004/10/15.

[SGHS98] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 231–242, July 1998.

[THM+03] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects, 2003.

[Ves04] Marcus Vesterlund. mzlair.se, 2004. http://www.mzlair.se/sphproject.en.php visited 2004/10/11.

[VRL] Research on complex materials at vrlab, umeå university. http://www.vrlab.umu.se/research/complex_materials.shtml visited 2004/12/02.

[WH94] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics*, 28(Annual Conference Series):269–277, 1994. http://www-2.cs.cmu.edu/afs/cs/user/aw/www/pdf/particles-reprint.pdf visited 2004/09/21.

[Whi94] F.M. White. *Fluid Mechanics*, page 19. McGraw-Hill, third edition, 1994.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, pages 227–234, aug 1986.

[ZPBG01] M. Zwicker, H. Pfister, J. Van Baar, and M. Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, pages 371–378, July 2001.

A Kernel derivates

Here follows the derivates of the smoothing kernels in use.

$$\nabla W_{\text{poly6}}(\mathbf{r}, h) = \frac{945}{32\pi h^9} \begin{cases} (h^2 - r^2)^2 \mathbf{r} & 0 \leq r \leq h, r = |\mathbf{r}| \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{41}$$

$$\nabla^2 W_{\text{poly6}}(\mathbf{r}, h) = \frac{945}{32\pi h^9} \begin{cases} (h^2 - r^2)(7r^2 - 3h^2) & 0 \leq r \leq h, r = |\mathbf{r}| \\ 0 & \text{otherwise} \end{cases} \tag{42}$$

$$\nabla W_{\text{spiky}}(\mathbf{r}, h) = \frac{45}{\pi h^6} \begin{cases} \left(\frac{h^2 + r^2}{r} - 2h\right) \mathbf{r} & 0 < r \leq h, r = |\mathbf{r}| \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{43}$$

$$\nabla^2 W_{\text{viscosity}}(\mathbf{r}, h) = \frac{45}{\pi h^6} \begin{cases} (h - r) & 0 \leq r \leq h, r = |\mathbf{r}| \\ 0 & \text{otherwise} \end{cases} \tag{44}$$

B Quick math reference

Some mathematic definitions as a quick reference:

Gradient:

$$\nabla f(x, y, z) = \frac{df}{dx}\mathbf{i} + \frac{df}{dy}\mathbf{j} + \frac{df}{dz}\mathbf{k} \tag{45}$$

Laplacian:

$$\nabla^2 f(x, y, z) = \frac{d^2 f}{dx^2} + \frac{d^2 f}{dy^2} + \frac{d^2 f}{dz^2} \tag{46}$$

Convolution:

$$(f \otimes g)(\mathbf{x}) = \int_{\mathbf{r}} f(\mathbf{r}) g(\mathbf{x} + \mathbf{r}) d\mathbf{r} \tag{47}$$