Fluid Simulation Using Implicit Particles

Advanced Game Programming

Dan Englesson
Joakim Kilby
Joel Ek

December 20, 2011

Abstract

This report covers the implementation of a fluid simulation using a hybrid method. The method used is heavily based on the contents of the book *Fluid Simulation for Computer Graphics* by Robert Bridson [1] and is commonly referred to as the PIC/FLIP method. In the method, the fluid surface is tracked using particles and its mass conserved by enforcing zero divergence on the deforming velocity field. We store the velocity field on a staggered grid as described in [2], which greatly helps to fulfill the mass-conserving criteria. We also reconstruct the actual fluid surface by evaluating the *Improved Blobbies* signed distance function introduced in [3] on a regular grid followed by a standard implementation of *Marching Cubes*.

We have tested our implementation using five different simulation cases and obtain great results, all of which are shown in the report together with benchmark data.

1 Introduction

Fire, water and air, once thought of as three of the four fundamental elements making up all known substances, seem to have more in common than once thought. All three are different types of fluids, or compounds which are able to flow and deform under the influence of commonly occurring forces such as gravity.

The complex behavior of fluids has always been fascinating to man and it is only in the latest decades that we have come to fully understand the complex equations governing their dynamic behavior. Fluid simulation is greatly useful since it can aid in the design of buildings and constructions but also since it allows one to simulate an event which never has to happen. For instance, the flooding of a city, a great explosion or anything else that would fit in a modern feature film.

2 Background

In order to understand how the different steps of the method work, some background information is needed. This section contains an explanation of the fundamental equations, a brief repetition of relevant vector calculus and a short description of the infamous *Stable Fluids* method. It also explains the difference between the Eulerian and the Lagrangian viewpoints as well as the benefits of using a staggered grid.

2.1 Governing equations

The two equations governing fluid flow are called the Navier-Stokes equations. The fundamental equation is given in equation 1 which describes how a velocity field V evolves over time. Naturally, the time derivative of this velocity field must be subject to external forces. This is modeled by the F term in equation 1 and is commonly representing gravity, wind or user interaction. The $\nu \nabla^2 V$ term is the diffusion term, which models the viscosity of the fluid.

Viscosity is a commonly used term in fluid simulation which simply means the tendency of the fluid to resist flow and is observed as its thickness. For example, water flows easily and therefore has low viscosity. Honey is thick and resists flow which is why its viscosity is high. The amount of viscosity is directly controlled by the scalar ν. It should be noted that this term is commonly discarded when simulating water. The amount of numerical dissipation introduced by interpolation errors is often a good replacement for this term, which otherwise would be solved by a Poisson equation.

$$\frac{\partial V}{\partial t} = F + \nu \nabla^2 V - V \cdot \nabla V - \frac{\nabla p}{\rho} \quad (1)$$

$$\nabla \cdot V = 0 \quad (2)$$

The $V \cdot \nabla V$ term is the self-advection term which models how the velocity field flows within itself. This can seem a bit strange but it is only natural as the velocity field represents the movements of the mass of a fluid. The final term, $\frac{\nabla p}{\rho}$, is the subtraction of the pressure gradient ∇p over the density ρ. When combining the entire equation with the constraint in equation 2, this term is replaced by a subtraction of a pseudo-pressure gradient. This as a result of any gradient of a scalar being curl-free and therefore removed when enforcing a divergence-free velocity field due to the Helmholtz-Hodge decomposition as stated in [4].

2.2 Representing fluids

There are two common ways of representing fluids which are based on completely different viewpoints. One way is to think of the fluid as a collection of tiny particles, or atoms, which together make up the volume of the fluid. This representation is a Lagrangian representation and has given rise to a whole family of purely

particle-based fluid simulations of which smoothed particle hydrodynamics, or SPH is the most common. These methods can be beneficial as their definition is intuitive and understandable but have a great flaw. The calculations are based on the assumption that the particle density always is sufficient. This is not the case as particles are able to spread out, allowing for regions with low density. In these regions, the calculations become less accurate as there simply is not enough information stored in the nearby particles.

Another way of representing fluids is from the Eulerian point of view. For this representation, the fluid is observed from a domain of interest which is divided into a number of cells. This forms a grid, which is why the Eulerian representation also is known as the grid-based representation. Grid-based representations do not suffer from the same problems with low-density regions as particle-based representations do. Instead, other problematic issues arise. In order for the fluid surface to be detailed enough, the grid must consist of a multitude of cells. It is not uncommon for purely grid-based fluid simulations to use millions of cells. This effect is due to the Nyquist criterion which states that the sampling frequency of a signal must be twice that of the highest frequency component. In other words, the simulation is only guaranteed to capture surface details larger than the magnitude of two cell widths. This is why the number of cells must be extremely large for purely grid-based methods.

A fluid simulation can also use a hybrid method, which combines the surface capturing possibilities of a particle-based method and couples it with an auxiliary grid to enforce accurate mass-conservation. Such a method is explained in detail in the method section.

2.3 Operators and vector fields

In order to understand how the velocity field is evolved and how the mass-conserving constraint is maintained, the following three operators need to be understood. Equation 3 shows the gradient operator. It operates on a scalar field and yields a vector field with each of the partial derivatives as its components. When the operator is applied to the cells of a grid structure, it should be interpreted as the local change of the scalar field q. In other words, the exchange of the property described by the scalar q.

$$\nabla q = \left(\frac{\partial q}{\partial x}, \frac{\partial q}{\partial y}, \frac{\partial q}{\partial z} \right) \tag{3}$$

Equation 4 shows the divergence operator. Divergence is related to the gradient operator and involves the same partial derivatives but instead of measuring each derivative separately, it measures the total local change. As shown below, it operates on a vector field and yields a scalar field. The operator should be interpreted as the total increase or decrease of any property being transported by the vector field $\vec{u}$.

$$\nabla \cdot \vec{u} = \frac{\partial \vec{u}}{\partial x} + \frac{\partial \vec{u}}{\partial y} + \frac{\partial \vec{u}}{\partial z} \tag{4}$$

The Laplace operator is the most complicated of the three operators discussed here. It can be seen in equation 5 and operates on a scalar field q. Even though the expression involves partial derivatives of the second order which might seem complex, it is actually the divergence operator applied to the gradient of a scalar field. In other words, a concise way of stating a sequential application of the two operators previously defined. When applied to the cells of a grid structure, it should be interpreted as the total interchange of material between all adjacent cells.

$$\nabla^2 q = \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} + \frac{\partial^2 q}{\partial z^2} \tag{5}$$

2.4 The staggered grid

In order to apply the continuous operators to a discrete grid structure, the operators

first need to be differentiated. This requires an introduction to a special type of grid, the staggered marker and cell (MAC) grid, introduced by Foster and Metaxas in [2]. The principal difference compared to a common regular grid is that the components of the velocity field are separated and offset. This is shown in figure 1 which shows that the sample points of the velocity field coincide with the cell face boundaries. Please note that all scalars which are to be transported by the velocity field have their sample points located at the center of a cell. This is important for the definition of the differentiated operators.

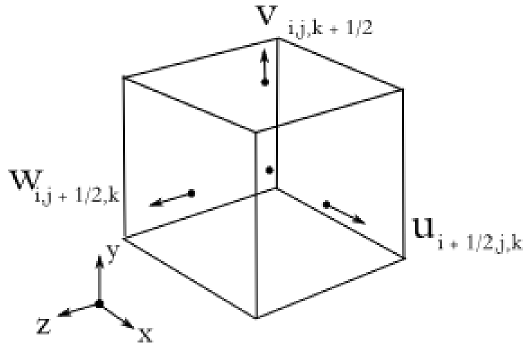

Figure 1: The staggered grid

The benefit of using this structure is that the evaluation of the discrete counterparts of the operators previously defined becomes greatly simplified. The structure introduces the curious half index for which a positive half index is the sample value stored at the face shared between a cell and its consequent cell.

Consider the discrete gradient shown in equation 6 where s is the size of a cell. The gradient is applied to the scalar q, which is stored in the center of the cells. The evaluation position must therefore be exactly between two adjacent cells. Please note that this evaluation is made componentwise and results in three different scalars, each with different evaluation positions. Also note that the evaluation positions coincide with the sample locations of the velocity field components.

$$\begin{aligned} (\nabla q)_{i+1/2,j,k} &= \left(\frac{q_{i+1,j,k}-q_{i,j,k}}{s}\right) \\ (\nabla q)_{i,j+1/2,k} &= \left(\frac{q_{i,j+1,k}-q_{i,j,k}}{s}\right) \\ (\nabla q)_{i,j,k+1/2} &= \left(\frac{q_{i,j,k+1}-q_{i,j,k}}{s}\right) \end{aligned} \tag{6}$$

The definition of the discrete divergence operator is shown in equation 7. As with the discrete gradient, s denotes the cell size. Divergence operates on component pairs of the velocity field and its evaluation position must therefore be the in the center of a cell.

$$\begin{aligned} (\nabla \cdot \vec{u})_{i,j,k} &= \frac{\vec{u}_{i+1/2,j,k}-\vec{u}_{i-1/2,j,k}}{s} \\ &+ \frac{\vec{u}_{i,j+1/2,k}-\vec{u}_{i,j-1/2,k}}{s} \\ &+ \frac{\vec{u}_{i,j,k+1/2}-\vec{u}_{i,j,k-1/2}}{s} \end{aligned} \tag{7}$$

The discrete Laplace operator is as stated a combination of the gradient and the divergence operator. In order to understand how it is defined on the staggered grid, first consider the application of the gradient operator. As previously stated, the evaluation position of the gradient is between two adjacent cells. If the divergence operator is applied to those six scalars, we obtain equation 8. As two operators are applied in effect, the scaling factor of s^2 is natural.

$$\begin{aligned} (\nabla^2 q)_{i,j,k} &= \frac{q_{i+1,j,k}+q_{i-1,j,k}}{s^2} \\ &+ \frac{q_{i,j+1,k}+q_{i,j-1,k}}{s^2} \\ &+ \frac{q_{i,j,k+1}+q_{i,j,k-1}}{s^2} \\ &- \frac{6q_{i,j,k}}{s^2} \end{aligned} \tag{8}$$

Please note that both the discrete Laplace operator and the discrete divergence operator are evaluated at the cell center. Since Poisson equations have the form shown in equation 9, using a staggered grid is clearly beneficial for solving these equations. Especially since it allows for the evaluation of central differences using a width of only a single cell size.

$$\nabla \cdot \vec{u} = \nabla^2 q \tag{9}$$

2.5 Stable Fluids

The Stable Fluids method was introduced by Jos Stam in 1999 [4]. It was the first unconditionally stable method for fluid simulation and introduced the concept of semi-Lagrangian advection. The Stable Fluids method calculates an approximate solution to the Navier-Stokes equations; equation 1 and 2.

A solution to equation 1 is obtained by sequentially calculating the contribution from each part of the equation, where for each step the input is given by the output of the previous step.

In order to enforce the constraint given by equation 2, the final velocity field obtained in the solution method is projected onto its divergence-free part.

The entire process is illustrated in figure 2 where $\vec{w_0}$ is the velocity field from the previous time step and $\vec{w_4}$ is the final, divergence-free, velocity field.

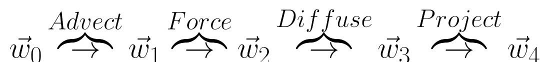

Figure 2: The Stable Fluids method

3 Method

3.1 Mesh conversion

In order to represent polygon mesh objects as fluids or solid objects for simulation purposes, a conversion must be made from polygons into a voxel representation.

To achieve this, a polygon object is read from an *.obj* file and as each triangle of the object is processed, its vertex coordinates are transformed into discrete grid coordinates. The voxel corresponding to those grid coordinates, the voxel in which the vertex lies, is then set as the cell-type the object is to represent, e.g. fluid or solid.

To avoid problems with triangles that are significantly larger than the grid-resolution, a number of points are selected, in a random fashion, on the triangle surface and transformed into grid coordinates as well.

This method creates a voxelized *shell*, or surface, resembling the original polygon object. However for simulation, a surface is insufficient and the voxelized model must be filled to create a volume.

The volume is created by filling out the gaps between voxels in the representation. This is achieved by selecting a voxel which is marked with the correct cell-type and then stepping along the x, y and z directions in turn until another voxel is found which has the same cell-type. All voxels between the two are then set as the correct cell-type.

This method works under the assumption that all polygon objects are *Manifold* surfaces; there can be no holes in the mesh, no self-intersection of polygons and no interior shells within the mesh.

3.2 Hybrid particle methods

There exist two main approaches for simulating fluid effects in computer graphics as mentioned earlier, namely the use of Lagrangian particles or Eulerian grids. The two approaches have their strengths and weaknesses, the Lagrangian particles are very good with the advection part but have problem with the pressure and incompressibility constraint. Fortunately, the Eulerian grid approach is excellent in solving the pressure and incompressibility constraint but due to interpolation errors from the semi-Lagrangian advection, the method has problem with the advection part. One can see that where the Lagrangian approach has its difficulties, the Eulerian approach is very good and by combining these two methods by letting Lagrangian particles handle the advection part and the Eulerian grid handle the pressure and incompressibility constraint, a better simulation of water effects can be achieved.

Many different hybrid approaches exists such as particle level sets, The Particle in Cell (PIC) and the Fluid Implicit Parti-

cle (FLIP) method, where the two latter methods will be further discussed in this paper. The particle level set method was introduced by Foster and Fedkiw in [5] and the PIC method was introduced as early as 1963 by Harlow in [6] and was later improved by Brackbill and Ruppel [7] with the FLIP method in 1986. The FLIP method where then introduced to incompressible flow by Zhu and Bridson in 2005 [3] and is today a state-of-the-art technique for fluid simulation.

3.2.1 The PIC/FLIP method

Even though the PIC and FLIP methods are basically the same method but with a slight difference in the velocity update, the fluid characteristics differ quite a bit. A fluid that only uses PIC is more viscous than a FLIP fluid, which is due to numerical dissipation because of the double interpolation of the velocity, but more on that in a bit. The FLIP method however, has little viscosity and is therefore very well-suited for water effects but unwanted visible noise on the surface is present. By linearly combining the PIC and FLIP methods, one obtain a fluid which has little viscosity and is free of surface noise.

The severe numerical dissipation of the PIC method is caused by interpolation. The particle velocities are transferred to the grid through interpolation which introduces some smoothing and then the smoothed velocity values are being used to solve the new velocities with the Navier-Stokes equations and are then interpolated back to the particles replacing the old velocities. As mentioned, due to this excessive double interpolation, a PIC fluid will appear more viscous than a FLIP fluid. Brackbill and Ruppel solved this problem by instead of interpolating the newly calculated velocities to the particles from the grid and replacing the velocities, they interpolated the change in velocity and added it to the already existing particle velocities. By doing so, only smoothing from one interpolation is performed and thus the smoothing is not accumulated as in the PIC method and makes the FLIP method almost free of numerical dissipation. Below follows a stepwise method-of-solution for a PIC/FLIP fluid simulation and a more in-depth view of the different steps.

1. Initialize the grid and the particle positions and velocities.

2. Transfer particle velocities to a staggered grid.

3. FLIP: Save a copy of the grid velocities.

4. Calculate and apply external forces.

5. Enforce the Dirichlet boundary condition.

6. Classify all voxels as fluid, solid or air.

7. Calculate the pseudo-pressure gradient using a Preconditioned Conjugate Gradient method.

8. FLIP: Update particle velocities by subtracting the new grid velocities from the saved grid velocities, interpolate and add the difference to the particle velocities.

9. PIC: Update particle velocities by interpolating and replacing the old velocities with the new grid velocities to the particles.

10. PIC/FLIP: Update the particle velocities by taking the interpolated FLIP and PIC velocities from 8. and 9. and for each particle weigh the PIC and FLIP velocities with a factor α between zero and one.

11. Update the particle positions with an ODE solver with the newly created velocities.

3.2.2 Initialize particles

Every voxel that is classified as fluid will have seeded particles inside. The particles are seeded in a jittered grid, much like super-sampling in a renderer, where eight particles in each fluid voxel is jittered by randomly placing the particles in a 2x2x2 sub-grid. A higher number of particles can be used in each voxel but does not necessarily give significantly better results but might instead slow down the calculations. Bridson stated in [3] that eight particles per fluid cell was enough to yield good results.

3.2.3 Transfer particles to grid

Since the particles are randomly placed in space and the calculations of the pressure and incompressibility are performed on the grid, some kind of interpolation is needed to transfer the nearby particle velocities to the grid. The grid velocities are updated from the nearby particles through trilinear interpolation of the weighted average particle velocities that lies in a cube twice the grid cell width where the center is at the grid-velocity component.

3.2.4 Calculate external forces

The external forces such as gravity and forces from any user interaction are added to the velocities by simple Euler integration as seen in equation 10, where V stands for velocity, F is the external forces and Δt is the time step.

$$V_{new} = V_{old} + F\Delta t \tag{10}$$

A simple Euler integration is enough since this task is unconditionally stable for reasonably small Δt. This is because there exists no feedback loop between the external forces and the velocity field.

3.2.5 Enforce Dirichlet boundary condition

The Dirichlet boundary condition states that there should be no flow into or out of solid cells to which has the normal n, see equation 11 where V is the velocity in the fluid cell and n is the normal to the neighboring solid.

$$V \cdot n = 0 \tag{11}$$

If a fluid cell has a solid neighboring cell, the velocity components are checked and if any of the velocity components point towards a neighboring solid cell, the velocity is projected to go along with the surface of the solid cell. Since the simulation is performed on a staggerd MAC-grid and the velocities are divided into separate components, the projection of the velocity is very simple. One just has to set it to zero if the component points into a neighboring solid cell.

3.2.6 Classify voxels

Firstly, the type of every non-solid voxel is cleared and set to *air*. Then voxels containing one or more particles are set to *fluid*. Solid voxels are set to be solid from the beginning of the simulation and and do not change their type.

3.2.7 Conserving mass

A mass-conserving velocity field is synonymous to a divergence-free velocity field. Conserving the mass of a fluid is therefore equal to enforcing zero divergence on the velocity field transporting mass.

This part of a simulation step is the most computationally heavy one as it involves solving a Poisson equation. Recall the previously stated Laplace operator and how it was defined as a summation of the partial derivatives of the second order. It should be emphasized that there is a dependency between every pair of two adjacent cells and that the solution to a Poisson equation in effect means solving a massive system of linear equations. Before this task can be undertaken, a derivation of how the Poisson equation is obtained is required.

The Helmholtz-Hodge decomposition states that any vector field can be expressed as a divergence-free part V_{df} and a curl-free part V_{cf} as shown in equation 12.

$$V = V_{df} + V_{cf} \tag{12}$$

Furthermore, vector calculus states that the gradient of any scalar fields is, by definition, curl-free. We can therefore replace the curl-free part of our velocity field with the gradient of an unknown scalar field, ∇q. This is shown in equation 13.

$$V = V_{df} + \nabla q \tag{13}$$

As with any equation, applying an operator to both sides of the equality sign does not change equality. Thus, it is allowed to apply the divergence operator to both sides. Since divergence is a linear operator, $\nabla \cdot (V_{df} + \nabla q)$ is equal to $\nabla \cdot V_{df} + \nabla \cdot \nabla q$ and we get equation 14.

$$\nabla \cdot V = \nabla \cdot V_{df} + \nabla \cdot \nabla q \tag{14}$$

Equation 14 can be simplified as we in the decomposition step defined V_{df} as being divergence-free. The divergence operator applied to V_{df} must therefore be equal to zero. Also, since we defined the Laplace operator as the consequent application of a gradient operator and a divergence operator, we get equation 15.

$$\nabla \cdot V = \nabla^2 q \tag{15}$$

Equation 15 is a Poisson equation in which q is the unknown scalar field which solves the equation. By rearranging equation 13 we obtain equation 16 which clearly shows that we can enforce mass-conservation on the velocity field by finding the unknown scalar field q and subtracting its gradient, ∇q.

$$V_{df} = V - \nabla q \tag{16}$$

Equation 15 is the equation which needs to be solved in order to find the scalar field q, often referred to as the pseudo-pressure. The left-hand side of the equation can be calculated simply by evaluating the divergence of the velocity field. The right-hand side contains the unknown scalar field and the Laplace operator. The definition of the operator is shown in equation 8 and states that adjacent values of the unknown scalar field should be used in the calculations. Naturally, this is not possible as the scalar field is unknown. However, the coefficients are fully known as they only depend on the local configuration of a cell. That is, if there are solid cells directly adjacent or if the fluid is free to exchange material through every cell face.

The definition of the Laplace operator becomes somewhat different for boundary fluid cells as the Dirichlet boundary condition states that there should be no flow through solid boundaries. The result on the Laplace operator is that the coefficient for the solid cell (when considering the adjacent fluid cell) is set to zero and that the central coefficient is increased by one.

As the coefficients are calculated and stored, a massive system of linear equations is obtained. The system is on the form $Ax = b$ where A is the coefficient matrix and b is the divergence of every cell. The number of equations in this system is directly related to the number of cells in the simulation grid. If there are $w \cdot h \cdot d$ cells, the size of the coefficient matrix is $(w \cdot h \cdot d)^2$.

Since there can be at most six other cells adjacent to every cell, this system is very sparse and can be solved efficiently by iterative methods taking advantage of this property. The memory requirement of the sparse coefficient matrix is $w \cdot h \cdot d \cdot 4$. This can be compared with the virtual size of the matrix previously stated.

In order to solve the Poisson equation, Bridson [1] recommended the preconditioned conjugate gradient method with a preconditioner of the modified incomplete Cholesky factorization type. This method has fast convergence and is able to operate on a sparse system. An implementation

of the PCG method was made with the pseudo-code in Bridson's book as a guide. Though it should be noted that any iterative method that does not require an explicit representation of the A matrix could be used, such as the *Jacobi iterative technique*.

When the Poisson equation is solved, the gradient of the resulting scalar field is subtracted from the velocity field and mass-conservation is guaranteed.

3.2.8 Update particle velocities

The particles need to be updated with the newly calculated velocities which are stored on the MAC-grid. This is done by trilinearly interpolating the velocities of the eight neighboring grid-velocities to the particle and, for PIC, update the velocity with the new velocity or for FLIP, interpolate the change in velocity and add it to the existing particle velocity. A linear combination of both PIC and FLIP can be used to get a low viscosity, water-like, fluid with no surface noise. This can be seen in equation 17 where $\vec{u}_p^{new}$ is the new particle velocity and α is the PIC blending factor. The factor determines how much PIC, viscosity or numerical dissipation there should be where 1.0 is pure PIC and 0.0 is pure FLIP. The $lerp()$ functions represents the trilinear interpolation function.

$$\begin{aligned}\vec{u}_p^{new} &= \alpha \cdot lerp(\vec{u}_{grid}^{new}, \vec{x}_p) \\ &+ (1-\alpha)[\vec{u}_p^{old} + lerp(\Delta\vec{u}_{grid}, \vec{x}_p)]\end{aligned} \quad (17)$$

3.2.9 The CFL condition

The CFL condition states that the a particle should always move less than one grid-cell in each sub step. It is done by taking the cell-width and dividing it by the maximum velocity in the grid to get a *stabledt*. The *stabledt* is then compared to the actual time step dt and if it is larger than dt, the *stabledt* is set to dt. The particles are then advected in six sub steps until it has reached dt. It is common to have around five sub steps.

3.2.10 Update particle positions

The particles positions are advected with a Runge Kutta 2 ODE solver, which is stable as opposed to a simple Euler ODE solver. However, the particles can occasionally penetrate solid boundaries due to errors in the RK2 solver. This can cause the particle that has penetrated a solid voxel to become stuck. To fix this problem, the particles that have penetrated a solid voxel are moved back in the normal direction to half the cell-width outside of the solid voxel.

3.3 Intermediate storage

The fluid solver is divided into two different parts; a simulation program which is responsible for all calculations regarding the movement of the fluid and a visualization program which takes simulation data as input and produces *.obj* files, meshes, from the data. The visualization program also has the capability to directly visualize particles or surfaces.

As the simulation program progresses, it sequentially writes the positions of the particles for the current time step into a binary file. The reason for writing only particle positions comes from the fact that the number of particles are very large and that the positions are all that is necessary to reconstruct surfaces as they determine the position of the fluid.

The position of a particle is represented by three floating point numbers, the Cartesian coordinates x, y and z. As each floating point number occupies four bytes of memory, the position of a particle occupies 12 bytes.

Table 2, in appendix A, shows the size of the simulation data for a few different amounts of particles, 250 frames are saved in each of the files.

3.4 Surface reconstruction

When a simulation has been performed and stored on disk, the surface must be reconstructed in order to be rendered in a standard rendering pipeline. As most rendering pipelines are optimized to handle triangles, this is often reduced to triangulating the simulation data. This is no easy task as there are no trivial solutions to the problem. Surface reconstruction is commonly achieved in two distinct passes. First, a scalar field is initialized from the underlying data on a regular grid using a function. The only requirement on this function is that it maps the data set onto a real-valued scalar. Note that real-valued implies that the function can assume both positive and negative values and that scalar means single value. In the second pass, a certain value (an iso-level) of the calculated scalar field is chosen to be visualized. The reconstructed surface is created where the function assumes this value.

3.4.1 The signed distance function

When dealing with point clouds, the mapping function commonly measures distance to the nearest point or average distance to some of the nearest points. Zhu and Bridson introduced a method called *Improved Blobbies* in [3]. The method calculates, for every grid corner, the average position and radius of the nearby points and determines whether the grid corner is located within the average radius of the average position. This gives negative distances for corners that are close to many points and positive distances for those who have few points nearby. In effect, this forms a signed distance function for which the definition can be seen in equation 18. However, when there are no points present within the fixed search radius, the function is undefined and this can cause problems.

The function is defined at every grid corner, which is why it makes sense to evaluate it for every grid corner. However, this is highly inefficient as the time complexity for such an implementation is $O(m \cdot n)$ where n is the number of points and m the number of grid corners. For optimization, we set the search radius R equal to three times the the grid spacing, the particle radius r_i to half the grid spacing and iterate through the particles. This makes the influence of every point easy to determine as it only can affect the corners located within its proximity. The time complexity is thus reduced from $O(m \cdot n)$ to $O(n)$.

$$\phi(\vec{x}_g) = len(\vec{x}_g - \sum_i w_i \vec{x}_i) - \sum_i w_i r_i \quad (18)$$

The points are weighted as shown in equation 19 using a well-shaped kernel function, shown in equation 20. The kernel function decays as the distance between the point and the grid corner grows, being exactly zero at a distance equal to the search radius, R. Every weight is normalized with respect to all contributions from nearby points. Implementing this weighting functionality is therefore most effectively done by accumulating contributions and their kernel values followed by a final normalizing step.

$$w_i = \frac{k_1(len(\vec{x}_g - \vec{x}_i)/R)}{\sum_j k_1(len(\vec{x}_g - \vec{x}_j)/R)} \quad (19)$$

$$k_1(s) = max(0, (1 - s^2)^3) \quad (20)$$

Since the kernel squares its input variable and the input always is the norm of a vector, we replace this kernel with an optimization, avoiding square roots. The improved kernel is shown in equation 22 and the modified weight function in equation 21. Note that the distance between $\vec{x}_g$ and $\vec{x}_i$ is calculated as the square distance, or dot product, which does not require a square root operation.

$$w_i = \frac{k_2((\vec{x}_g - \vec{x}_i) \cdot (\vec{x}_g - \vec{x}_i)/R^2)}{\sum_j k_2((\vec{x}_g - \vec{x}_j) \cdot (\vec{x}_g - \vec{x}_j)/R^2)} \quad (21)$$

$$k_2(s) = max(0, (1-s)^3) \tag{22}$$

3.4.2 Marching Cubes

A simple and elegant surface reconstruction algorithm was introduced by Lorensen and Cline in [8]. The new method went under the name *Marching Cubes* and has since become the industry standard for iso-surface generation. The method is based on evaluating a real-valued function into a discrete scalar field defined on a regular grid. It should be noted that this grid is different from the one used during the simulation pass. They are not required to be of the same resolution, yet it is implied that this grid should be of a resolution determined by the density of the tracker particles in the simulation. If eight particles are seeded per fluid cell, the optimal resolution for the surface reconstruction grid will be that of the simulation grid according to the Nyquist criterion.

The first step of the method is to evaluate the signed distance function for every cell corner in the grid. Since our signed distance function is not defined everywhere in the simulation domain, special consideration has to be made about where to evaluate the function. Cell corners where the function is undefined is flagged as outside for the following steps of the algorithm.

Following the first step is to, given a certain iso-value, classify cell corners as either inside or outside of the surface. Since we are using a signed distance function which represents the signed distance to the surface, the iso-value of interest is zero. Determining which cell corners that are outside thus corresponds to testing whether the value is greater than zero and vice versa.

Depending on how the corners are classed as either inside or outside, a configuration is formed for every cell. This configuration needs to be enumerated. Since there are eight corners of a cell, there

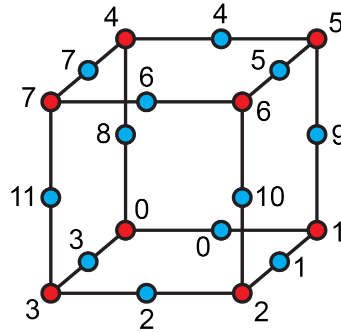

Figure 3: The corner and edge numbering

can be exactly 256 different combinations, each forming a unique case. The configuration is perfect for storage in a single unsigned byte which can be used as an index into a case table, detailing which triangles that form the surface. The notation shown in figure 3 is used to encode a one for every corner that is flagged as inside.

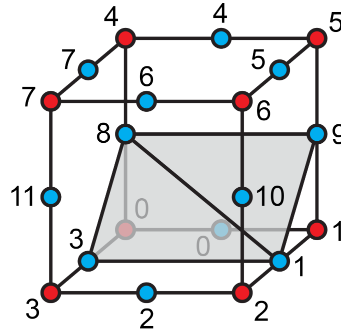

Figure 4: A unique case

For cells where only corners 0 and 1 are inside, the corresponding index will be 00000011 where the least significant bit is located to the right. This is equal to 3 and thus case 3 is to be used from the triangle case table. From the standard implementation of Marching Cubes, this corresponds to placing two triangles. One between vertices 3, 1 and 8 and one between vertices 8, 1 and 9. This case is visible in figure 4 which clearly shows that a surface is constructed which encapsulates corners 0 and 1. The actual positions of

the vertices are determined by interpolating the scalar values located at the corners as shown in equation 23 where α is the blending factor between vertex i and vertex j.

$$\alpha = \frac{0 - \phi_i}{\phi_j - \phi_i} \tag{23}$$

The original Marching Cubes algorithm also details how the vertex normals can be calculated from the values in the scalar field. The gradient is simply evaluated at every grid corner and interpolated to the vertex positions. However, this is also problematic as the function can be undefined outside of the surface. Instead surface normals are summed at the each vertex and finally normalized. As the vertex normals are calculated from the topology of the surface rather than from the gradient of the scalar field, the resulting quality of the vertex normals is lower. The two calculation methods are not identical yet both produce vertex normals of sufficient quality for our purposes.

It should be noted that the original Marching Cubes algorithm can create ambiguous cases for which the generated surface will have holes. The method used in [9] details how these cases can be found and corrected properly.

3.5 Exporting meshes

During the surface reconstruction pass, a vertex list, a vertex normal list and a triangle list is created. Vertices pending addition to the vertex list are compared to the existing vertices. If they are located within a small threshold value (10^{-5}), the old vertex is reused and triangles are therefore able to share vertices. This reduces the number of vertices in the model and can also be used to disable the creation of the unwanted tiny stretched triangles, simply by raising the threshold value. It should be noted that our implementation uses a brute force method which has a time complexity of $O(n)$ in which every new vertex is compared to the already existing ones. An efficient implementation should make use of a data structure to reduce the time complexity of this operation.

From these lists, the surface must be stored in a highly accessible and standardized format. We chose the object file format (.obj) from Alias Wavefront since it is text-based and easy to implement. The format is also compliant with the indexed face set data structure, which consists of the lists previously described.

4 Results

Our implementation was tested using five different test cases. Test cases one and two were designed to show how the viscosity of the simulated fluid can be changed simply by altering the PIC influence factor. All simulated cases are shown in table 1, where additional simulation data are shown in 2 in appending A.

Case	Description
1	Dam break (FLIP)
2	Dam break (PIC)
3	Dam break with dragon
4	Falling cube
5	Splashing dragon

Table 1: List of simulation cases

All images shown below in appendix B, were rendered in Autodesk 3DSMAX using simulation data from our implementation. For each case, a set of 250 frames were simulated, reconstructed and rendered.

5 Discussion

We have found that the use of hybrid methods as opposed to pure Eulerian methods for fluid simulation allows for the capture of highly detailed behavior in the fluid. Our method is not limited by the grid resolution since the fundamental representation of the fluid is the set of particles. This allows us to perform

simulations at a relatively low grid resolution while still producing detailed surfaces, saving both time and memory. Our method also captures the detail in splashes with greater accuracy than Eulerian methods.

There is however a drawback to our method; the surface reconstruction from a set of particles, e.g. a point cloud, is harder to perform than that of a level set and the complexity of the generated surfaces create a larger number of polygons. The complexity of the surface can be somewhat simplified by setting a limit on the number of particles that must be present in a voxel to generate a surface. This will however reduce the detail in fine features such as splashes and a trade-off must be made between detail and complexity.

Our implementation is able to perform simulations in a relatively short time even as the number of particles increase. This is mainly due to clever data structures and a mindfulness of how the cache memory and caching works. As table 2 clearly shows, the bulk of the time it takes to produce a frame is spent on rendering and surface reconstruction. We have noted that as the complexity of our simulation data increases, e.g. when large splashes occur, the reconstruction phase takes longer and produces more complex geometry with a larger amount of polygons. This is something which we plan to remedy and a possible solution is mentioned in the next section.

6 Improvements & future work

A major area of improvement in our implementation is the surface generation. A first step would be to change the way in which a particle may influence the grid points. In the current implementation, a particle influences grid points within a sphere of a certain radius from its location. This *sphere-of-influence* could be changed into a more elliptical shape with an orientation and size based on the local particle density. The most crucial improvement this would yield is the ability to define sharper features in the surface. As it stands today, extremely sharp features are smoothed out because of the spherical shape of influence from a particle. This is described in more detail in [10].

We would also like to introduce post-processing operations on the generated surfaces, mainly to reduce the number of unnecessary polygons and by doing so decreasing the rendering time. Amongst the operations we would like to implement are;

Mesh-smoothing to reduce noise-like behavior which may be introduced by low local particle density.

Decimation simply to reduce the number of polygons. We believe that mesh decimation based on polygon area and *curvature* would be very effective in reducing the amount of polygons whilst preserving the fine detail in the mesh.

As for the simulation part of our implementation, we would like to introduce seeding of *foam particles* at voxels where the magnitude of the *curl* is above a certain threshold to emphasize the wild behavior of the fluid. We would also like to introduce a *two-way coupling* in our simulator, meaning that the fluid may interact with solid objects, or other fluids, in a more complex way, e.g. having floating solid objects or blending of oil and water.

We would also like to improve the grid structure so that we can represent solid objects which have a slope in a correct way. Using the existing structure to do this would result in a staircase-like behavior as a result of the voxel representation. Bridson has suggested a way of achieving this in [1].

Finally we have considered implementing an *adaptive grid* structure. In this structure, the grid would only be defined in the close vicinity to where the fluid actually is and it would follow the fluid as it moves. We believe that this would reduce

the simulation time significantly.

References

[1] R. Bridson, *Fluid Simulation for Computer Graphics.* A K Peters/CRC Press, Sept. 2008.

[2] N. Foster and D. Metaxas, "Realistic animation of liquids," in *Graphical Models and Image Processing*, pp. 23–30, 1995.

[3] Y. Zhu and R. Bridson, "Animating sand as a fluid," in *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, (New York, NY, USA), pp. 965–972, ACM, 2005.

[4] J. Stam, "Stable fluids," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, (New York, NY, USA), pp. 121–128, ACM Press/Addison-Wesley Publishing Co., 1999.

[5] N. Foster and R. Fedkiw, "Practical animation of liquids," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, (New York, NY, USA), pp. 23–30, ACM, 2001.

[6] M. Evans and F. Harlow, *The particle-in-cell method for hydrodynamics calculations.* LA-2139, 1957.

[7] J. Brackbill and H. Ruppel, "FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions," *Journal of Computational Physics*, vol. 65, pp. 314–343, Aug. 1986.

[8] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, pp. 163–169, August 1987.

[9] T. S. Newman and H. Yi, "A survey of the marching cubes al-

gorithm," *Computers & Graphics*, vol. 30, pp. 854–879, Oct. 2006.

[10] J. Yu and G. Turk, "Reconstructing surfaces of particle-based fluids using anisotropic kernels," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, (Aire-la-Ville, Switzerland, Switzerland), pp. 217–225, Eurographics Association, 2010.

A Benchmarks

	1	2	3	4	5
Simulation grid	128x64x64	128x64x64	128x64x64	128x64x128	128x64x64
Particle density	2^3	2^3	2^3	2^3	4^3
Particles	917 600	917 600	595 200	438 976	402 880
PIC influence	5%	45%	5%	5%	5%
Simulation	34 min	18 min	25 min	8 min	7 min
Reconstruction	82 min	64 min	71 min	155 min	36 min
Rendering	483 min	396 min	310 min	926 min	366 min
Simulation data	2.56 GB	2.56GB	1.66 GB	1.22 GB	1.12 GB
Surface data	820 MB	709 MB	829 MB	1.34 GB	572 MB

Table 2: Benchmark data for all simulation cases

B Images

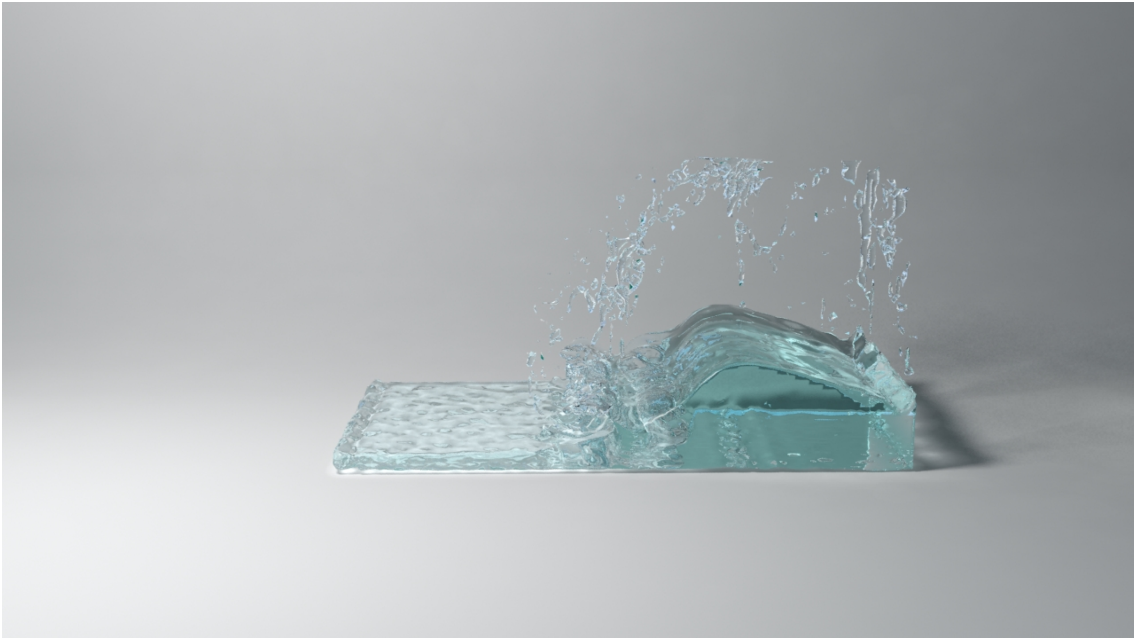

Figure 5: Dam break (FLIP)

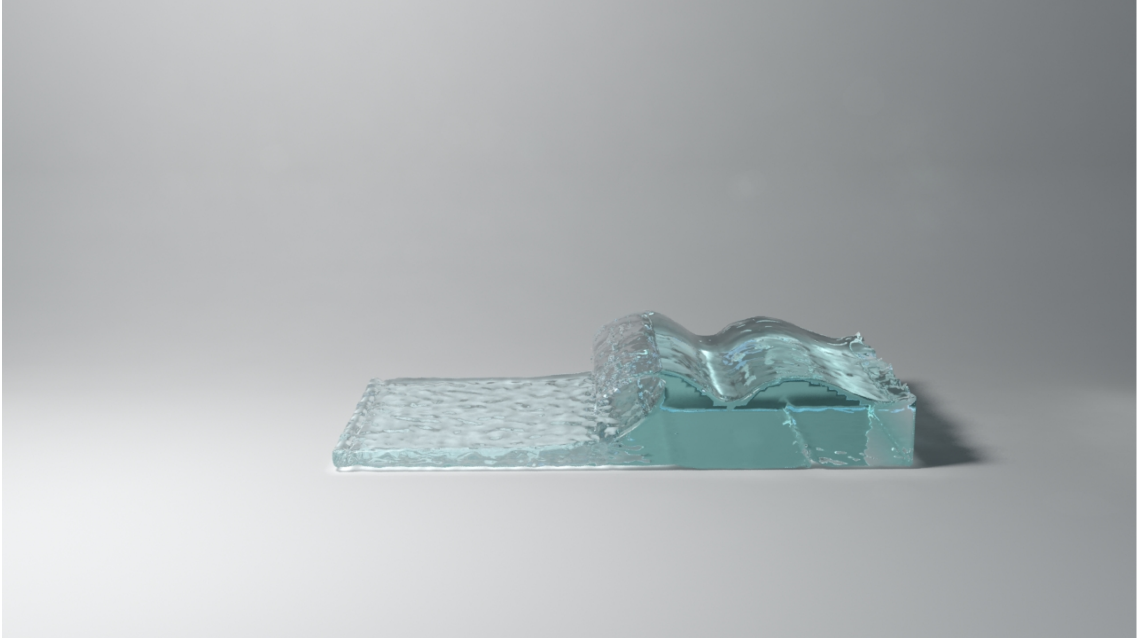

Figure 6: Dam break (PIC)

Figure 7: Dam break with dragon

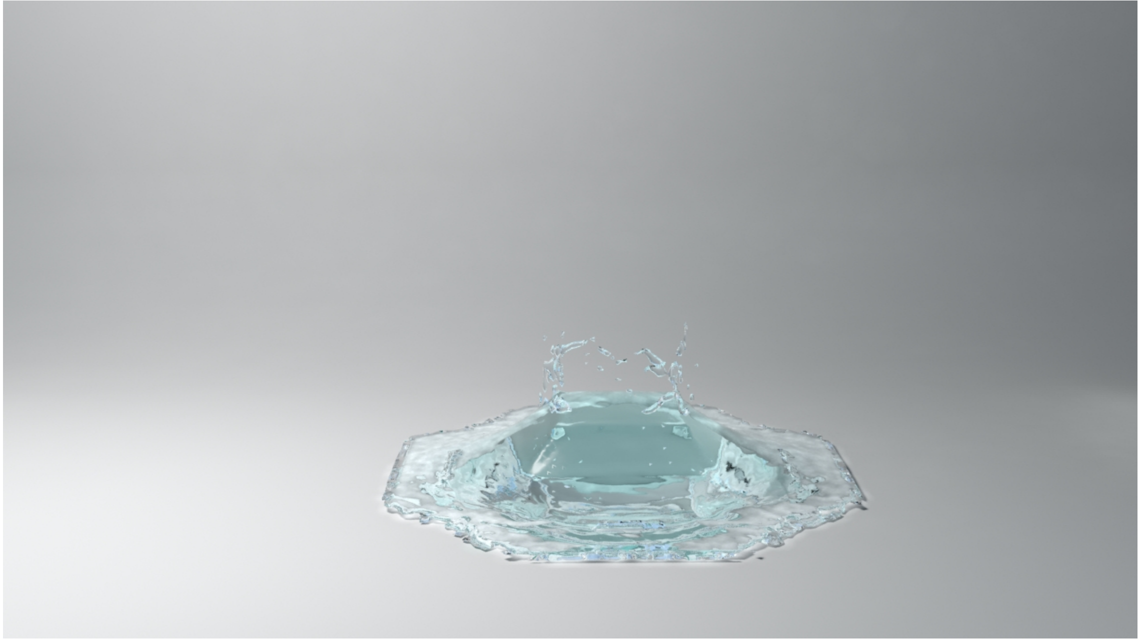

Figure 8: Falling cube

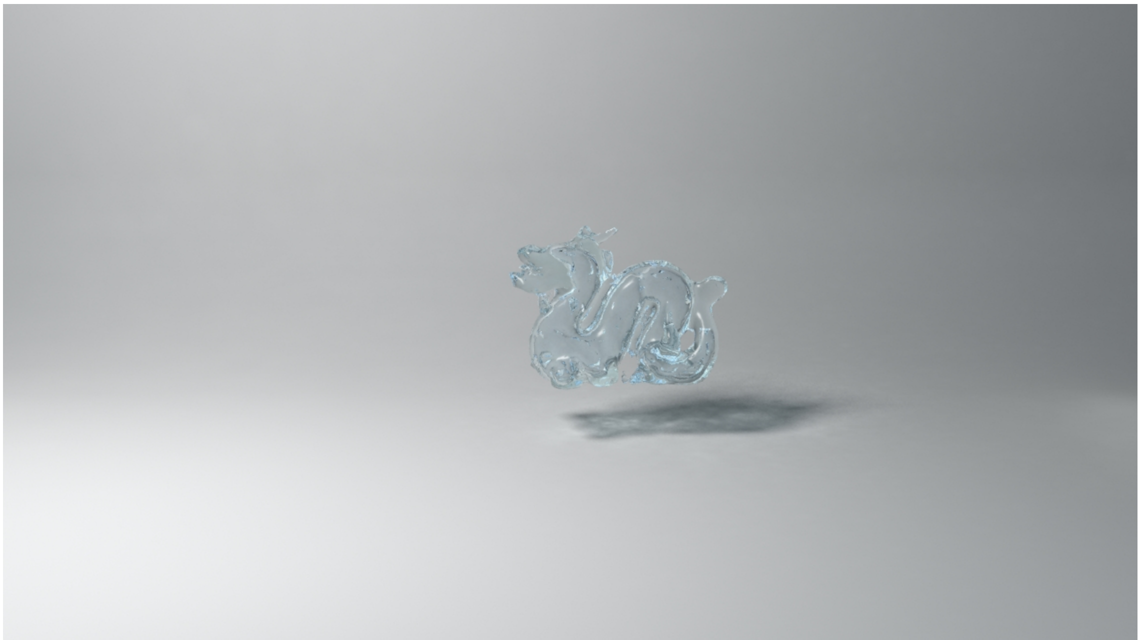

Figure 9: Splashing dragon