3D Real-Time Fluid Simulation

grid-based approach with marker particles

Alexander Sommer
Hochschule RheinMain
Unter den Eichen 5
65195 Wiesbaden, Germany
alex-sommer@gmx.de

ABSTRACT

This work shows how to implement a 3D grid based fluid simulator with a simple visualization. It explains the details of the implementation of a preconditioned incomplete cholesky conjugate gradient (PICCG) algorithm as a pressure solver. Furthermore it shows how to use marker particles in a hybrid blend of PIC and FLIP techniques to track the movement of the fluid.

Keywords

Fluids; Physically Based Animation; 3D Fluid Simulation; Real-Time; Grid Based; Eulerian Approach; PCG; MIC(0); PICCG; Marker Particles; PIC; FLIP;

1. INTRODUCTION

Fluids are per definition substances with a continual flow under a shear stress. We are surrounded by fluids every day. They can be classified as compressible and incompressible. Compressible fluids, like gases, can significantly change their density. On the other hand, incompressible fluids like water will keep a near constant density. There are no perfect incompressible fluids, but the change of density in the example of water is so little that we can treat them as incompressible with a constant density. Since fluids play such a big role in our daily life, it seems obvious that we want to understand and simulate the behavior of fluids. In computer graphics, the animation of the phenomena of different fluids play an important role. Our work aims to implement a three dimensional real-time simulation of incompressible fluids.

2. MATHEMATICAL BACKGROUND

As already mentioned in section 1 fluids can be divided into two categories: compressible and incompressible. Incompressible fluids can be described by the **incompressible Navier Stokes Equations** (1) & (2).

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} + \frac{1}{\rho}\nabla p = \vec{F} + \nu \nabla \cdot \nabla \vec{u} \tag{1}$$

Table 1: Symboles in Navier Stokes Equations

Symbol	Meaning
$\vec{u}$	velocity vector
t	time
ρ	density
p	pressure
$\vec{F}$	body forces
ν	viscosity coefficient

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

They form a set of partial differential equations. Since we're more focused on the implementation of the fluid solver, we will skip the derivation of these equations, but they can be found in several physics textbooks like **A Mathematical Introduction to Fluid Mechanics**[3]. An explanation on the meaning of the variables in these equations can be found in table 2. It's worthwhile to mention that it can be useful to use the real physical units instead of using dimensionless values, when it comes to tweaking the behavior later.[2]

For the rest of this work the viscosity term will be neglected, making these equations simpler and easier to use, while still providing useable results. The results of this approximation are called **Euler Equations** (3) & (4).[2]

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = \vec{F} \tag{3}$$

$$\nabla \cdot \vec{u} = 0 \tag{4}$$

In the following the body force $\vec{F}$ will be the gravitation ($9.81\frac{m}{s^2}$ in negative y-direction) and we'll be using the density of water $\rho = 1000\frac{kg}{m^3}$, to simulate water-like behavior.

3. GRID BASED VS. PARTICLE BASED APPROACH

We can distinguish between two different approaches when tracking the movement of a continuum. The first and more intuitive one is the Lagrangian approach, where we have particles with a position $\vec{x}$ and a velocity $\vec{v}$ representing parts of the continuum, like molecules in a fluid. The other approach is the Eulerian. Here, we look at fixed locations in space and see how different quantities like velocity and temperature change in time at these locations. This is a grid-based approach. Both techniques have their justification and are being used for different purposes. In fluid simulation, the grid-based approach is preferable, because it's easier to calculate differences like the pressure gradient from a fixed grid than from a cloud of moving particles, as we will see shortly.[2]

In the following sections, we will focus on the implementation of a grid-based fluid solver, where we calculate all quantities of the fluid on discrete grid points. Later we will introduce marker particles, a particle-based approach to track the movement of the fluid through the discrete field, combining particle-based techniques on our grid-based fluid simulation.

4. THE GRID

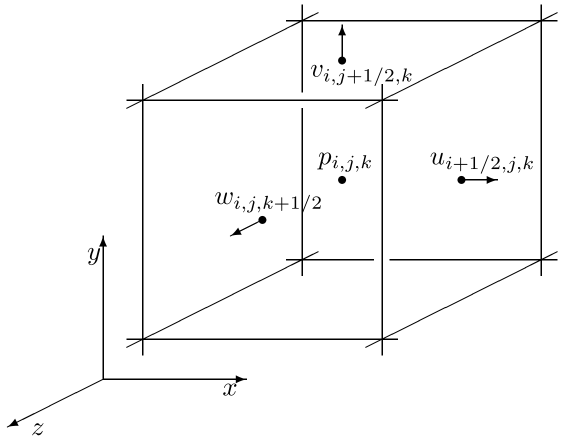

Figure 1: Three Dimensional MAC Cell

As already discussed in section 3, we will focus on the implementation of a grid-based simulation. It's necessary now to define the structure of the grid that we'll be using to track the movement of the fluid through the volume. Probably the most common standard is the **Marker-and-Cell** (MAC) grid technique proposed by Harlow and Welch[5]. It's a staggered grid, since it stores the different quantities at different locations on the grid. The pressure value is stored at the center of every grid cell, while the velocity values are stored at the faces of the cell. It should be mentioned that the velocity values are the normal component of the velocity at each cell face. A three-dimensional illustration of the grid can be seen in figure 1. Arranging the grid in this way brings a huge accuracy advantage when it comes to estimating the derivative at a sample point i.

$$\frac{\partial q}{\partial x}_i \approx \frac{q_{i+\frac{1}{2}} - q_{i-\frac{1}{2}}}{2\Delta x} \tag{5}$$

When calculating the derivative with this staggered central difference the formula achieves $O(\Delta x^2)$ accuracy, while forward or backward differences would only achieve $O(\Delta x)$ accuracy. On the flip-side, every time we want to know a pressure value offside an actual grid point, a trilinear interpolation is required. For a velocity value even three trilinear interpolations, for the three components of the velocity in 3D, need to be done.

5. ALGORITHM

The whole algorithm we'll be using for implementing a grid-based fluid simulation can be divided into four sub-algorithms (see Code 1). In the following sub-sections we will take a closer look at each of these algorithms.

Code 1: Grid-Based Fluid Simulation Algorithm

```
start with divergence free field $\vec{u}^0$
for each time $t_n$
   Choosing a Timestep: $\Delta t$ for $t_n \rightarrow t_{n+1}$
   Advection: $\vec{u}^A = advect(\vec{u}^n, \Delta t, \vec{u}^n)$
   Applying Body Forces: $\vec{u}^B = \vec{u}^A + \Delta t\vec{F}$
   Pressure Projection: $\vec{u}^{n+1} = project(\Delta t, \vec{u}^B)$
```

5.1 Choosing a Timestep

As we've seen in code 1, we would like to know the state of the fluid at m different times. So, what we need to calculate is the transition between a time n and a time $n+1$. Often this time difference is too big to be done by only one calculation without losing numerical accuracy. We need to find a timestep Δt as big as possible without losing numerical accuracy. The advection algorithm presented soon is unconditionally stable, as we will see, since we're only interpolating from values on the grid. This means we can't create bigger or smaller values that will blow up the simulation. But for the project algorithm we need this limitation in the timestep. A good rule of thumb timestep size is:[2]

$$\Delta t \leq \frac{5\Delta x}{u_{max}} \tag{6}$$

With u_{max} being the biggest velocity value in the grid and Δx the cellsize of the grid.

5.2 Advection

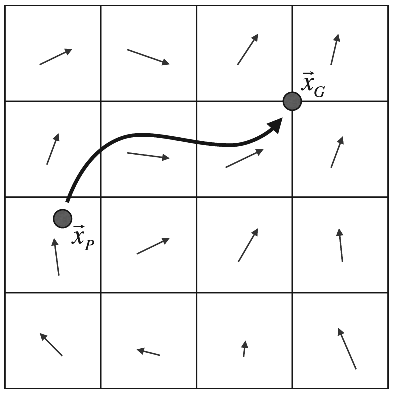

Figure 2: semi-Lagrangian method for virtual particle backtracing

In this chapter we will talk about the advection algorithm.

$$q^{n+1} = advect(\vec{u}, \Delta t, q^n) \tag{7}$$

Basically what we would like to know is how a given property q at a grid point (i,j) changes from time step n to timestep $n+1$. To get this new value we will be using a technique called **semi-Lagrangian method**. We basically track back the virtual particle that is at timestep $n+1$ at grid point (i,j,k). It should be mentioned that we're not actually creating any particles, we just use them as theoretical construct for tracing back our needed properties. By the use of simple Forward Euler in backwards time (8), we get its position p on the grid at timestep n (see fig. 2 for a 2D illustration).

$$\vec{x_p} = \vec{x_G} - \Delta t \vec{u}(\vec{x_G}) \tag{8}$$

It should be mentioned that for a better accuracy a Runge-Kutta Method (like RK3) instead of the Forward Euler Method should be used. We'll talk about that later, when we talk about marker particles. Since we know that these virtual particles don't change their properties q, we know that the property q of the particle at position p in timestep n is equal to q at grid point (i, j, k) in timestep $n + 1$. The problem that comes up is that p is usually not directly on the grid, so we need to interpolate this property from the ones at the surrounding grid points. The easiest way to interpolate this is by doing a trilinear interpolation.

For better accuracy, it's recommended to do at least a tricubic interpolation. A problem we will encounter is what to do when the virtual particle's location is not inside the fluid domain, where we have no information about the property q on the grid to interpolate from. We'll introduce a way of extending properties in unknown domains by **breadth-first search** later in the section about marker particles.

For the basic fluid simulation we want to implement, the only property which needs to be advected is the velocity itself. We need keep in mind that the velocity value, as mentioned earlier, is stored at the faces of the grid and not in the center of the cells. So when it comes to advecting a u value (x-component of $\vec{u}$), we first need to find the total velocity $\vec{u}$ at the location where u is stored by interpolating between the surrounding v and w values. Same for advecting v and w values.

5.3 Applying Body Forces

$$\vec{u}^B = \vec{u}^A + \Delta t \vec{F} \tag{9}$$

In one timestep Δt all external forces $\vec{F}$, so called **body forces**, result in an additional change of the velocity field. Recall some basic physical knowledge a force is simply a mass times an acceleration. This acceleration changes the velocity field and we have to calculate these forces in (9). Without a force the fluid will not move. In the most basic case the one body force we always have to deal with is gravitation.

$$\vec{F} = \vec{g} = (0, -9.81\frac{m}{s^2}, 0) \tag{10}$$

For animation purposes, it's possible to add translucent forces as body forces to achieve certain effects. For this basic implementation, we'll just stick to gravitation.

5.4 Pressure Projection

$$\vec{u}^{n+1} = project(\vec{u}^B, \Delta t) \tag{11}$$

By now we learned how to propagate properties q through the velocity field in a timestep Δt. Recall the only property that needed to be advected was the velocity itself, we called this $\vec{u}^A$. We applied the body forces, in our case gravity, to the preliminary vector field $\vec{u}^A$ and called this $\vec{u}^B$. What we want to do now is to apply a pressure gradient, coming from new to be calculated pressure values, to this vector field to receive the final vector field $\vec{u}^{n+1}$at time $n + 1$.

$$\vec{u}^{n+1} = \vec{u}^B - \Delta t \frac{1}{\rho} \nabla p \tag{12}$$

This field should be divergence free so it satisfies the incompressibility condition,

$$\nabla \vec{u}^{n+1} = 0 \tag{13}$$

while also satisfying the boundary conditions.

$$\vec{u}^{n+1} \vec{n} = \vec{u}_{solid} \vec{n} \quad \text{at solid boundaries} \tag{14}$$

$$p = 0 \quad \text{at free surfaces} \tag{15}$$

This step is the heart of every fluid simulation.

5.4.1 *Setting up a system of equations*

First let's take a look at formula (12) again and substitute the pressure gradient with central differences.

$$u^{n+1}_{i+1/2,j,k} = u_{i+1/2,j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \tag{16}$$

If the voxel (i, j, k) contains fluid then the pressure $p[i, j, k]$ is unkown and what we would like to solve for it. If the voxel (i, j, k) is empty we simply set the pressure $p[i, j, k] = 0$. This is a direct boundary condition or **Dirichlet boundary condition**. It gets more complicated if voxel (i, j, k) is solid, then the pressure $p[i, j, k]$ has no valid value. What we want to do here is to use a ghost value for the pressure at this voxel, that satisfies the boundary condition at solid boundaries (see formula (14)).

$$p^{ghost}_{i+1,j,k} = p_{i+1,j,k} - \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j,k} - u^{solid}_{i+1/2,j,k}) \tag{17}$$

This second type boundary condition is called a **Neumann boundary condition**.
Now let's take a closer look at the incompressibility condition (formula (13)). We can easily calculate the divergence by using central differences

$$(\nabla \vec{u})_{i,j,k} = \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} \tag{18}$$

If we now substitute the pressure-gradient formulas (16) into the incompressibility condition (18) we get a linear equation for each fluid grid cell with unknown pressures.

$$\frac{\Delta t}{\rho} \left(\frac{6pi_{i,j,k} - p_{i+1,j,k} - p_{i,j+1,k} - p_{i,j,k+1} - p_{i-1,j,k} - p_{i,j-1,k} - p_{i,j,k-1}}{\Delta x^2} \right) = -\frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} - \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} - \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} \tag{19}$$

Keep in mind that we aren't setting up equations for empty cells or cells containing solid, but we might refer to their pressure value. For these cells we substitute the pressure as mentioned above.

5.4.2 Converting to Matrix-Vector Form

When we take a look at the equation for the unknown pressure (19), we see that if we want to calculate the pressure $p[i,j,k]$ of cell (i,j,k) we need to know the pressure of the surrounding cells, making these equations a coupled system, that can only be solved as a whole. It makes sense to write this linear equation system in standard matrix-vector form.

$$Ap = b \tag{20}$$

Where b are the negative divergences (18), p is the variables vector and A the coefficient matrix. Let's take a closer look at matrix A. As we can see in the left-hand side of (19) each row of the coefficient matrix A has at most seven entries other than zero. This is the case if every neighboring cell of the cell corresponding to this row contains fluid. Otherwise that row has even less entries. This creates a matrix containing mostly zeros, a so-called **sparse matrix**. It's obvious that storing all these zeros would be very inefficient. We have to come up with a better way of storing this kind of matrix. But first take another look at the properties of A. A is a symmetrical matrix, meaning $A_{(a,b)} = A_{(b,a)}$ or in the three indices notation $A_{(i,j,k),(i-1,j,k)} = A_{(i-1,j,k),(i,j,k)}$. This means we only have to store half of the off-diagonal values of A. One method of using as little as possible memory to store the entries of A is to store the diagonal values in one array $A_{(i,j,k),(i,j,k)} = \texttt{Adiag[i,j,k]}$ and the entries of the neighboring cells in positive direction in three more arrays $A_{(i,j,k),(i+1,j,k)} = \texttt{Ax[i,j,k]}$, $A_{(i,j,k),(i,j+1,k)} = \texttt{Ay[i,j,k]}$ and $A_{(i,j,k),(i,j,k+1)} = \texttt{Az[i,j,k]}$. If we need to access a neighboring cell in negative direction we can use the symmetry property and refer to it as $A_{(i,j,k),(i-1,j,k)} = A_{(i-1,j,k),(i,j,k)} = \texttt{Ax[i-1,j,k]}$.

5.4.3 Solving the linear equation system

Matrix A is a special type of matrix, a so called **seven-point Laplacian matrix**. There are several techniques to solve linear equation systems with this type of matrix. The technique we will be using to solve this system is called **Modified Incomplete Cholesky Conjugate Gradient, Level Zero**, short **MICCG(0)**.
First, let's take a look again at the characteristics of A. As we've already discussed A is a symmetric matrix, further more we can say that A is **positive semi-definite**, meaning that $q^TAq \geq 0$ for any arbitrary non zero vector q. In the usual case A is even strictly positive definite $q^TAq > 0$, meaning also that all of its eigenvalues are positive. Conjugate gradient (CG) algorithms are in general a good way to solve symmetric positive semi-definite linear systems. CG algorithms start with an initial guess at the solution and converge iterative towards the correct solution. The problem with the CG algorithm is that the farther A is from being the identity matrix the longer the iteration process will take. We will add a preconditioner to counteract this problematic. Roughly the idea is that we can add a second invertible matrix M from the left so that $MAp = Mb$. If M is approximately the inverse A^{-1} of A, then MA is pretty close to being the identity matrix, which will lead to a fast convergence of the CG algorithm. That means we have to come up with a fast way for computing M, while being as close to A^{-1} as possible. We will use a modified version of the **Incomplete Cholesky** (IC) method for this preconditioning process. If we recall our memory about solving linear systems we remember that we can split up a matrix A into a lower- and upper triangle matrix. For a symmetric matrix, these two are the transposes of each other.

$$A = LL^T \tag{21}$$

This split up is called Cholesky factorization, and we can solve the original system $Ap = L(L^Tp) = b$ by solving first $Lq = b$ and then $L^Tp = q$. The problem with this splitting up is that it is likely that L has way less non zero entries than A, making the solver easily run out of memory for bigger grids. The idea of IC is every time we create an entry in L that is a zero in A we keep it zero in L too. We get a lower-triangular matrix L with the same non-zero pattern as A so $LL^T \approx A$. We can come up with an even more efficient way of storing the entries of L. Instead of splitting up A into lower-triangle we can explicitly distinguish between strict lower triangle F and diagonal D

$$A = F + D + F^T \tag{22}$$

It can be shown that for a matrix A of this particular form we can write L as

$$L = FE^{-1} + E \tag{23}$$

Where E is a diagonal matrix. This comes with the big benefit that we only need to calculate the diagonal entries of L, while reading the rest of the entries of L directly from A

$$E_{i,j,k} = \sqrt{\begin{array}{c} A_{(i,j,k),(i,j,k)} - \left(A_{(i-1,j,k),(i,j,k)}/E_{(i-1,j,k)}\right)^2 \\ - \left(A_{(i,j-1,k),(i,j,k)}/E_{(i,j-1,k)}\right)^2 \\ - \left(A_{(i,j,k-1),(i,j,k)}/E_{(i,j,k-1)}\right)^2 \end{array}} \tag{24}$$

It should be mentioned that all entries in E that are referring to a cell not lying on the grid or not containing a fluid should be zero. We will tweak the calculation of E slightly, taking account of unwanted non-zeros in the diagonal. This is called **Modified Incomplete Cholesky**.

$$E_{i,j} = \sqrt{\begin{array}{c} A_{(i,j,k),(i,j,k)} - \left(A_{(i-1,j,k),(i,j,k)}/E_{(i-1,j,k)}\right)^2 \\ - \left(A_{(i,j-1,k),(i,j,k)}/E_{(i,j-1,k)}\right)^2 \\ - \left(A_{(i,j,k-1),(i,j,k)}/E_{(i,j,k-1)}\right)^2 \\ -A_{(i-1,j,k),(i,j,k)} \\ \cdot \left(A_{(i-1,j,k),(i-1,j+1,k)} + A_{(i-1,j,k),(i-1,j,k+1)}\right)/E^2_{i-1,j,k} \\ -A_{(i,j-1,k),(i,j,k)} \\ \cdot \left(A_{(i,j-1,k),(i+1,j-1,k)} + A_{(i,j-1,k),(i,j-1,k+1)}\right)/E^2_{i,j-1,k} \\ -A_{(i,j,k-1),(i,j,k)} \\ \cdot \left(A_{(i,j,k-1),(i+1,j,k-1)} + A_{(i,j,k-1),(i,j+1,k-1)}\right)/E^2_{i,j,k-1} \end{array}} \tag{25}$$

This will, without proof here, boost the speed of the solver vastly from $O(n)$ to $O(n^{1/2})$ for a grid size of n.[2]

Now we know how to calculate the preconditioner with the MIC(0) method. We pack this in a method to apply this so created matrix $M = LL^T$ to a vector $z = Mr$ by first solving $Lq = r$ and then $L^T z = q$. Now all that's left to talk about is how to solve with the conjugate gradient algorithm and this preconditioner for the pressure p.

Code 2: Conjugate Gradient algorithm with preconditioner

```
set initial guess p = 0
set residual vector r = b
  if r = 0 return p
set auxiliary vector z = applyPreconditioner(r)
set search vector s = z
$\sigma$ = dotproduct(z,r)
loop until done or maximum iterations reached
  z = applyA(s)
  $\alpha$ = $\sigma$ / dotproduct(z,s)
  p = p + $\alpha$s
  r = r - $\alpha$z
    if max(r) $\leq$ tol return p
  z = applyPreconditioner(r)
  $\sigma_{new}$ = dotproduct(z,r)
  $\beta$ = $\sigma_{new}$ / $\sigma$
  s = z + $\beta$s
  $\sigma$ = $\sigma_{new}$
return p
```

The algorithm to solve a linear system in form of $Ap = b$ with PCG is shown in code 2. Basically what happens is that we start from an initial guess for the pressure $p = 0$ and calculate for each iteration the difference from the correct solution, the residual:

$$r_i = b - Ap_i \tag{26}$$

Once this residual is small enough ($\leq tol$) we stop the iteration and return the last value for the pressure p. A good method to compare the vector $\vec{r}$ with a tolerance here is to use the infinity norm $\|r\|_\infty$. For the tolerance a value on the order of 10^{-6} is good to start with. It's useful to implement a maximum number of iterations before the algorithm stops to prevent from freezing the program. A good suggestion is a maximum number of iterations of 200. While it comes in handy for the rest of the simulation to use single precision numbers to keep the memory usage as small as possible, single precision round errors especially while calculating the dot-products can lead to a significant slower convergence. The suggestion is to use double precision numbers at least for calculating these scalars.[2]

Now we have the new pressure values for satisfying the boundary conditions and we can update the velocity values accordingly to the pressure gradient.

6. MARKER PARTICLES

By now we have discussed how to calculate all necessary properties of the fluid when going from a time n to a time $n + 1$. What we haven't talked about yet is how the fluid will change its shape accordingly to the changed properties, like floating into new domains or leaving old domains. So far, we've only calculated new quantities, which will end in an equilibrium when we don't update the flow of the fluid.

We will introduce marker particles [5], that represent the fluid. We will track their movement through the velocity field out of current fluid cells into empty cells. When after a time step a cell does not contain any marker particles anymore we will mark this cell as empty vice versa if an empty cell is populated by a marker particle we will mark this cell as a fluid cell. We will also have to deal with particles that travel into solid cells or leave the grid. We end up modifying the algorithm from section 5 (code 1) to what you can see in code 3. We will briefly discuss the new sub algorithms in the following.

Code 3: Fluid Simulation Algorithm with marker particles

```
Seed Particles in Fluid Cells
for each time $t_n$
  Update Labels
  Update Velocity Fields from Particles
  Applying Body Forces
  Pressure Projection
  Update Particle Velocity from Velocity Fields
  Advect Particles
```

6.1 Seed Particles

As mentioned earlier, we want to use the particles for tracking the movement of the fluid. That leaves the question how many particles we need for a good tracking since it's clearly not practical to track the movement of every fluid molecule. A good rule of thumb is to use eight marker particles per fluid cell.[2] After we've initially loaded the scene, we will place one particle in each octant of each fluid cell. We will add a random jitter to the location of each particle to prevent from generating strange looking artifacts or stripe-like patterns in certain situations.

6.2 Update Labels

After each time step, the position of the marker particles will have changed. What we want to do now is to update the labels of each grid cell. Every cell that is marked as a solid will stay a solid, every cell that is not a solid and contains a marker particle will be marked as fluid. Every other cell will be marked as empty.

6.3 Update Velocity Fields from Particles

This step is what we called in section 5 the velocity advection. Since we now have actual marker particles, we don't need a semi-Lagrangian backtracking of virtual particles, we can populate the velocity fields from the velocity values of the marker particles. This approach is called **Particle-In-Cell** (PIC) [4]. Recall that the marker particles are not lying exactly on the grid points so we need to interpolate the grid values from the values of the particles. The easiest way to do so is by summing up weighted particle values at each grid point.

$$u_{i,j,k} = \sum_p u_p \frac{k\left(\vec{x}_p - \vec{x}_{i,j,k}\right)}{W} \tag{27}$$

Where W is a normalization factor and k the weighting kernel:

$$k(x, y, z) = h\left(\frac{x}{\Delta x}\right) h\left(\frac{y}{\Delta x}\right) h\left(\frac{z}{\Delta x}\right) \tag{28}$$

$$h(r) = \begin{cases} 1-r & : 0 \le r \le 1 \\ 1+r & : -1 \le r < 0 \\ 0 & : \text{otherwise} \end{cases} \tag{29}$$

We end up with the velocity fields populated by values from the marker particles near the grid points. All entries of the velocity fields that couldn't get a value from a nearby particle will be marked as unknown. In the following we will describe a method to extrapolate the known field values into these unknown areas.

6.4 Extrapolation

The extrapolation of field values is not explicitly mentioned in code 3, but is essential to its success at two different positions. As we've seen before we'll only get field values near the marker particles, but for the calculation of the divergences in the pressure projection and later in the advection of the particles it is important to know how the field values are estimated in unknown areas.

In section 5.2, we have already explored a way of estimating unknown field values by interpolation, but we saw that it was quite painful to deal with boundary conditions when ending up inside solids. We will now use a method of extrapolation where we don't have to worry about boundary conditions anymore: **Breadth-First Search**. The algorithm for the extrapolation with breadth-first search is shown in code 4.

Code 4: Data Extrapolation with breadth-first search

```
data array $q_{i,j,k}$
marker array $m_{i,j,k} = 0$ for known values and MAX_INT
    for unknown values
wavefront list $W$ with grid indices $(i, j, k)$
//first wavefront
loop over the entire grid with $(i, j, k)$:
  if $m_{i,j,k} \neq 0$ but at least one neighbor has $m = 0$
    push $(i, j, k)$ onto $W$
//further wavefronts
set $t = 0$ //current wave
while $t <$ depth //desired extrapolation depth
  let $(i, j, k) = W[t]$
  set $q_{i,j,k} =$ average of neighbors $(i', j', k')$ of $(i, j, k)$
      where $m_{i',j',k'} < m_{i,j,k}$
  for any neighbor $(i', j', k')$ with $m_{i',j',k'} =$ MAX_INT
    set $m_{i',j',k'} = m_{i,j,k} + 1$
    push $(i', j', k')$ onto $W$
  set $t = t + 1$
```

6.5 Update Particle Velocity from Velocity Fields

With the updated velocity fields from applying body forces and pressure projection extrapolated throughout the whole grid with the breadth-first search, we're now ready to move the marker particles through the grid. But first we need to update the particle's velocity according to the new values in the field.

During velocity field update from the particles, we already talked about the PIC approach, meaning storing quantities like velocity directly in the particle instead of on the grid. We saw that the grid values where interpolated causing numerical dissipation. If we now transfer these in-precise values back to the particles, we end up with even more numerical dissipation. This can lead to smoothing and unwanted effects. To counteract this variation in PIC, the **Fluid Implicit Particle** (FLIP) method was developed [1]. In FLIP the change in the quantity is interpolated and will be used to increment the existing particle value, instead of just interpolating the new value. This means in this case we need to store the old velocity fields temporarily to interpolate the change. We end up with only one smoothing instead of two in PIC. FLIP is really effective in reducing numerical diffusion from advection but it may develop noise. As we have mentioned we'll have eight particles in the fluid cells, meaning we have more degrees of freedom in the particles than in the grid. This can lead to unwanted effects like vanishing velocity fluctuations. What we want to do is to blend in a little bit of the PIC update into the FLIP update to eliminate the factor of noise. We end up with the new particle velocity:

$$\vec{u}_p^{new} = \alpha\,\texttt{intp}(\vec{u}_{grid}^{new}, \vec{x}_p) + (1-\alpha)\left(\vec{u}_p^{old} + \texttt{intp}(\Delta\vec{u}_{grid}, \vec{x}_p)\right) \tag{30}$$

We can see that if $\alpha = 0$ then only the FLIP update is done and if $\alpha = 1$ only the PIC update is done. A rule of thumb in relation to the kinematic viscosity of the fluid ν is[2]:

$$\alpha = \frac{6\Delta t \nu}{(\Delta x)^2} \tag{31}$$

Since we neglected the viscosity term $\nu = 0$, we will chose a small $\alpha = 0.02$ to blend in a little of the PIC update to factor out noise.

6.6 Advect Particles

Since we now have stored the new velocities as a particle properties we're ready to move the particles through the field. As we've already mentioned in section 5.2 simple Forward Euler is pretty inaccurate, which wasn't that much of an issue, since the error was reset in each time step in the semi-Lagrangian method. For the marker particle advection the error will propagate, so we need to work with a higher precision. We will use a **three-stage third-order Runge-Kutta scheme** [6] to interpolate the values:

$$\vec{k}_1 = \vec{u}\,(\vec{x}_n) \tag{32}$$

$$\vec{k}_2 = \vec{u}\left(\vec{x}_n + \frac{1}{2}\Delta t \vec{k}_1\right) \tag{33}$$

$$\vec{k}_3 = \vec{u}\left(\vec{x}_n + \frac{3}{4}\Delta t \vec{k}_2\right) \tag{34}$$

$$\vec{x}_{n+1} = \vec{x}_n + \frac{2}{9}\Delta t \vec{k}_1 + \frac{3}{9}\Delta t \vec{k}_2 + \frac{4}{9}\Delta t \vec{k}_3 \tag{35}$$

It should be mentioned that after particle advection some particles may end up outside the grid or inside solids. We will simply drop particles outside the grid and try to bring particles inside solids back to the nearest fluid domain. If we fail to find the nearest fluid domain we will drop these particles too. If we allocate a nearest fluid cell, we will move the particle $\frac{\Delta x}{4}$ away from the border.

7. VISUALIZATION

By now we've discussed all the necessary mathematical operations needed to get the fluid in physically correct motion. What's left to do is to find a way to display the calculated results in a feasible way.

7.1 Display Particles

The simplest approach is to just draw all the marker particles as dots on an empty canvas. In our implementation we added lines to visualize the bounding box in which the fluid is held (see fig. 3). This representation is suitable for showing some phenomena like swirls and droplets, but it's not yet a closed surface visualization which you would imagine when you think of water.

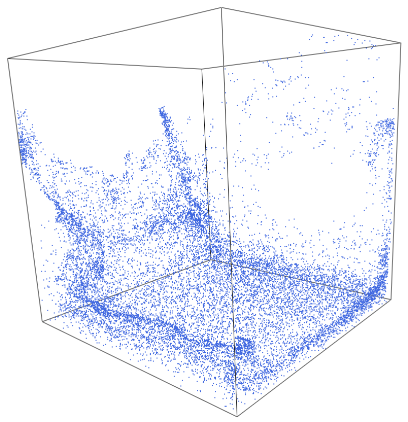

Figure 3: Visualization: drawing marker particles as dots

7.2 Marching Cubes

For creating a closed surface, we need a three dimensional contouring algorithm. We make use of the marching squares algorithm[8]. With the marching squares algorithm we create a triangle mesh of the isocontour where the pressure of the fluid is lower than a defined tolerance. Physically correct we would want to draw the surface where the fluids pressure is zero, so we would set the tolerance to zero. But we got a more pleasing and smooth look by raising the tolerance to a value between 5Pa and 20Pa.

We extract this surface by dividing the pressure grid into cubes, also called voxels. Each of these voxels has eight corners containing one pressure value from the grid. Each of these corners can be above the tolerance ($>$) or equal and lower than the tolerance ($\leq$). This makes it $2^8 = 256$ different configurations, also called cases, for each voxel. The cases define if and how the surface is drawn inside this voxel. There is a total number of 15 different cases (see fig. 4). Each other case is just a rotation or reflection of one of these cases.

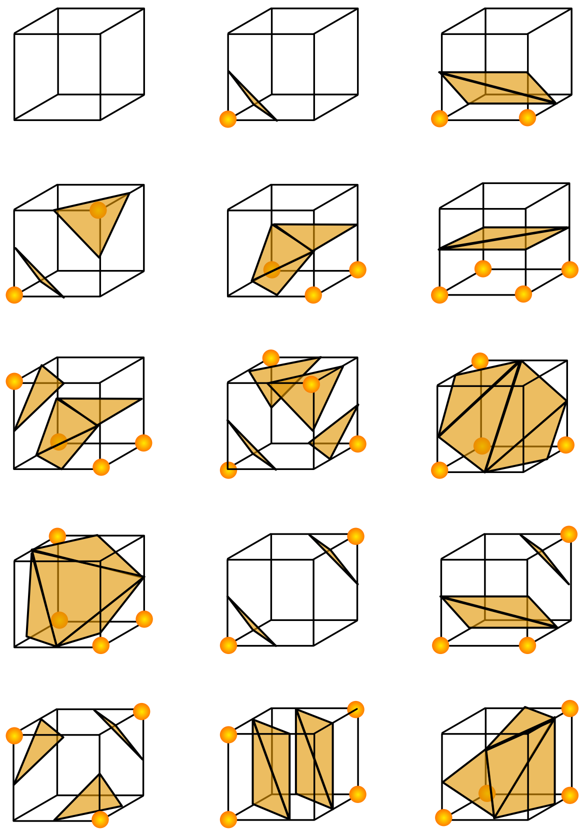

Figure 4: 15 differenct marching cubes cases, yellow corners are above tolerance

It should be mentioned that there are two ambiguous cases: case 11 (middle in the second last row of fig. 4) and case 14 (middle in the last row of fig. 4). These cases could be handled as break-cases separating the surface, as seen in figure 4, or as join cases joining the geometry. Furthermore some neighboring case configurations can lead to topology disfunctions creating artifacts and holes in the surface. Handling these issues requires further computations [7]. Since we're interested in a fast triangulation for a real-time simulation, we neglect these calculation enabling us to use a look-up table for the 256 cases. This may generate artifacts in some situations, but since each frame will only be shown for a very short period of the time, we accept this trade-off for a better performance.

As you can see in figure 6 (top) we're now able to see the fluid as a closed surface. On the flip-side we lost detail against the particle visualization. Furthermore since we're only able to see the ambient color of the surface we lost all perception of depth. The fluid surface looks like a flat two dimensional structure.

7.3 Shading

When we want to bring back depth into our flat looking surface, we need to introduce light generating highlights and shadows. We'll be using the Phong reflection model [9] for adding a diffuse and a specular component into our ambient color, generating a three dimensional look. With the Phong reflection model we're able to calculate the illumination I_p of each point on the surface.

$$I_p = c_a i_a + \sum_{l \in lights} \left(c_d(\vec{L}_l \cdot \vec{N}) i_{l,d} + c_s(\vec{R}_l \cdot \vec{V})^{\alpha} i_{l,s} \right) \quad (36)$$

Where i_a is the ambient illumination with the corresponding constant c_a, $i_{l,d}$ the diffuse and $i_{l,s}$ the specular illumination for each light source. α is the material dependend shininess constant, a higher value makes objects more mirror-like. See figure 5 for the vector relations in (36).

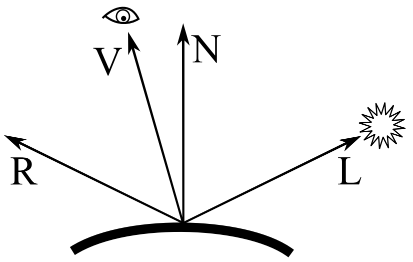

Figure 5: Vectors for calculating the Phong shading

What's left to discuss is how we get the surface normals needed for the illumination calculation. While it might sound like the logical approach to calculate a face normal per triangle, it's actually way better to calculate per vertex normals, since vertex normals can get extrapolated throughout the triangle at low cost by the GPU like other vertex attributes. When we calculated the marching cubes, we only looked a grid values at the corners of each voxel. When calculating normals we also need to count in neighboring voxels, otherwise we'll get two different normals for overlapping vertices in two different voxels. Each vertex is placed off grid in one axis. Let's assume this axis is the x-axis at $(i + 1/2, j, k)$. We know that between (i, j, k) and $(i+1, j, k)$ one grid value is above tolerance and one below or equal. Furthermore we need to check if there's also a tolerance crossing in the neighboring cells in y and z direction. We introduce a function checking if a grid value is above tolerance or not.

$$\texttt{aTol}(\texttt{grid}(\texttt{i},\texttt{j},\texttt{k})) = \begin{cases} 1 & : \text{grid(i,j,k) is above tolerance} \\ 0 & : \text{otherwise} \end{cases} \quad (37)$$

With this function we can calculate the normal in our example with:

$$v_x = \texttt{aTol}((\texttt{i},\texttt{j},\texttt{k})) - \texttt{aTol}((\texttt{i}+1,\texttt{j},\texttt{k})) \quad (38)$$

$$\begin{aligned} v_y &= \tfrac{1}{2}\left(\texttt{aTol}((\texttt{i},\texttt{j}-1,\texttt{k})) + \texttt{aTol}((\texttt{i}+1,\texttt{j}-1,\texttt{k}))\right) \\ &\quad - \tfrac{1}{2}\left(\texttt{aTol}((\texttt{i},\texttt{j}+1,\texttt{k})) + \texttt{aTol}((\texttt{i}+1,\texttt{j}+1,\texttt{k}))\right) \end{aligned} \quad (39)$$

$$\begin{aligned} v_z &= \tfrac{1}{2}\left(\texttt{aTol}((\texttt{i},\texttt{j},\texttt{k}-1)) + \texttt{aTol}((\texttt{i}+1,\texttt{j},\texttt{k}-1))\right) \\ &\quad - \tfrac{1}{2}\left(\texttt{aTol}((\texttt{i},\texttt{j},\texttt{k}+1)) + \texttt{aTol}((\texttt{i}+1,\texttt{j},\texttt{k}+1))\right) \end{aligned} \quad (40)$$

$$\vec{n} = \frac{\vec{v}}{|\vec{v}|} \quad (41)$$

The calculations for the vertices placed off grid in y- and z-axis are done similar. Now we're able to place a light in the scene and calculate the illumination model to get a three dimensional look on the fluid surface. Figure 6 (bottom) shows the result of the lighted scene in comparison to the flat surface in figure 6 (top).

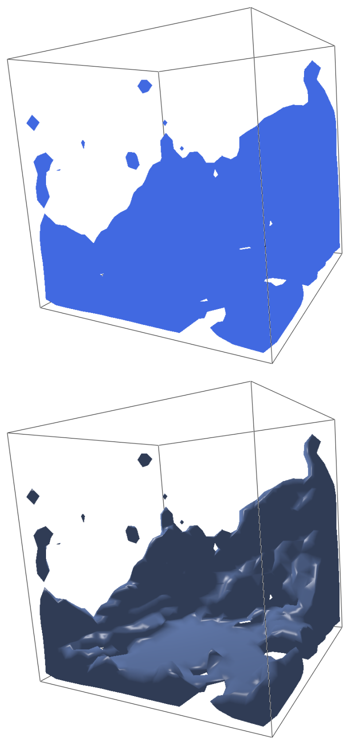

Figure 6: Top: Marching Cubes without lighting; Bottom: Marching Cubes with Phong lighting model

8. REFERENCES

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

[2] R. Bridson. *Fluid Simulation for Computer Graphics.* CRC Press, Boca Raton, FL, second edition, 2016.

[3] A. J. Chorin and J. Marsden. *A Mathematical Introduction to Fluid Mechanics.* Springer, New York, NY, 1993.

[4] F. H. Harlow. *The particle-in-cell method for numerical solution of problems in fluid dynamics.* Mar 1962.

[5] F. H. Harlow and J. E. Welch. Numerical calculation of time dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189, 1965.

[6] R. King. Runge-kutta methods with constrained minimum error bounds. *Mathematics of Computation*, 20(95):386–391, 1966.

[7] E. Lengyel. Transition cells for dynamic multiresolution marching cubes. *Journal of Graphics, GPU, and Game Tools*, 15(2):99–122, 2010.

[8] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, Aug. 1987.

[9] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.