Hybrid Particle-Grid Water Simulation using Multigrid Pressure Solver

Per Karlsson

February 2014

Abstract

This thesis involves an evaluation of the multigrid method for solving systems of differential equations in hybrid particle-grid fluid simulations. The work in this thesis is focused on inviscid incompressible liquid and water simulations and the method of choice is Fluid Implicit Particle (FLIP). Equations and algorithms are presented in but not restricted to a two-dimensional domain, and can easily be extended to three dimensions.

The implementation in this thesis is based on the Navier-Stokes fluid equations and the Level Set methods for surface tracking. The fundamentals of these equations and methods are explained in the first chapters and solutions are presented later on. A large portion of this thesis explains the multigrid method and its implementation for solving the pressure equations needed in a fluid simulation.

The results of the multigrid pressure solver in this thesis shows that the method is sufficient for non real-time simulations in computer graphics. A comparison between multigrid and the traditional preconditioned conjugate gradient method showed similar results in tests for correctness.

Contents

1 Introduction

In general, computer graphics tries to come up with methods to represent what we see and interpret with our eyes, where the main goal has always been to produce an image in the end that looks reasonable. There are two restrictions, time and accuracy. In most cases, we do not have unlimited time to produce each image, especially not in animation where there are 24 images per second that have to be constructed. The other limitation is accuracy. Some problems we do not know how to model and some problems we do not know how to solve precisely. Simplifications are accepted as long as the end result looks convincing.

In the field of animation, we care about how objects move in space and how they deform over time. Many problems can be solved by simply moving objects in space manually, often referred to as hand-animating. A common technique for animating characters in the visual effects and video games industry is to only animated a simplified skeleton and then weight geometry to different parts of the skeleton. This is fine for a characters main movement. However, for more complex things on characters, like hair and cloth, simple hand-animated approaches do not work anymore. There are too many hair strands or wrinkles on the cloth to control them all manually. Even if we would attempt to, it is very likely that the motion would look unreal. For more complex phenomena we need to simulate the objects and their geometry for a convincing motion.

This report focus on simulation of interesting water effects. Liquid fluids have, unlike their gas state, a surface defining their volume. The surface has a lot of degrees of freedom and can take any shape. To make it even more complex, it also changes topology over time which makes it hard to write surface tracking implementations. The other challenging part of fluid simulation is to define how the velocity in the fluid is changing over time where each little part of the fluid can have a velocity completely different from its neighbors.

1.1 Background

There have mainly been two different traditional ways of trying to solve the fluid equations, the Lagrangian viewpoint and the Eulerian viewpoint. The Lagrangian approach treats continuum like a particle system where each point in the fluid is labeled as a particle with a position and velocity. The Smoothed Particle Hydrodynamics[13], SPH, was introduced to computational fluid dynamics and animation by Desbrun and Cani[14]. The big advantage of SPH is that the advection step does not suffer from any numerical smoothing and act directly on the particles. It gets trickier to approximate spatial derivates since it completely depends on how many neighboring particles there are next to a particle. This can lead to instability in the simulation. Eulerian methods are grid based and instead of tracking a part of the fluid as it moves, they focus on tracking a fixed point in space and check how the fluid quantities at that point are changing over time. Foster and Metaxas[15] introduced the first 3D grid based water simulation. One problem grid based solutions had in the beginning was that they tended to blow up eventually and they were not stable for long simulations. J.Stam[5] added a semi-langrangian advection scheme to the grid based methods and made it unconditionally stable which allowed larger timesteps than before. To track surfaces, signed distance fields and level sets have been used. Fedkiw and Foster[6] introduced a way to combine particles and a level set for better surface tracking and therefore have less volume loss in the simulation. Unlike the Lagrangian approach, approximating a spatial derivate is a lot easier on a grid where one can just look at the difference between neighboring cells. Grid methods have many steps that requires interpolation, which tend to smooth out interesting high frequency motion in the fluid.

Both the Lagrangian and Eulerian have their advantages and disadvantages. Zhou and Bridson[2] came up with a way of combining them to have a stable pressure solving step on a grid and a Lagrangian advection step. This method is usually called *Fluid Implicit Particle*, FLIP. Zhou and Bridson used a preconditioned conjugate gradient for solving the pressure equations. In this report we are going to use the multigrid approach instead, inspired by the work of Mueller[3].

1.2 Purpose

The main goal in this thesis is to examine if it is possible to run the multigrid pressure solver method in a FLIP fluid simulation. More specifically, the type of fluid simulation being evaluated is going to be a water simulation. The domain of the simulation in this report is going to be two dimensional instead of three to make equations and explanations simpler. Although two dimensions, any method presented in thesis needs to have a corresponding three dimensional method that is straightforward and intuitive to implement, i.e. we cannot take advantage of being in a two dimensional environment that is less complex and use methods that do not scale well in three dimensions.

Speed is another important factor in this work. There are not going to be any constraints that requires the simulation to run in real-time but the computation time of a method is still going to be an important factor. The purpose of the simulation is to eventually end up in a video. To be practical, it is convenient if the simulation is fast. This decreases the time and effort an artist has to spend in order to complete a sequence with a fluid simulation in it.

The Lagrangian part of the FLIP method in this thesis is going to be based on the original computer graphics FLIP paper published by Zhou and Bridson[2]. The multigrid pressure solver discussed in this report is inspired by the multigrid implementation presented by Mueller[3].

Lastly, there will be restrictions to the grid dimensions of the fluid simulation and the boundary conditions of the fluid itself. To simplify the the multigrid method, the size of a dimension has to be a be a power of two. This is not a necessary condition but it will make it easier to explain the basics of a multigrid approach. When setting up the boundary conditions in the pressure equations, it will be assumed that the solid boundary walls are not moving over time, i.e. the velocity is zero. This assumption makes it easier to set up the pressure equations.

2 Fundamentals

This chapter will explain the major concepts that one needs to know about before attempting to implement a FLIP fluid solver. It will explain the theory behind the fluid equations and how to improve accuracy when dealing with discretization of differential equations. An overview of the overall FLIP algorithm is presented at the end of the chapter.

2.1 Navier-Stokes equations

Most fluid research in computer graphics is based on the famous incompressible Navier-Stokes equations. The equations can be written in different ways but in this report we will write them as

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

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

where $\vec{u}$ is the velocity field of the fluid, ρ is the density, p is the pressure, $\vec{g}$ represent all the external forces acting on the fluid (such as gravity) and υ is the viscosity constant. This report emphasizes on liquid simulations and in particular, water simulations. Water has by nature little vicosity. This means that the term in the Navier-Stokes equations that involves viscosity can be neglected as it is not as important as the other terms. If we drop the viscosity term in Equation 1, we get

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

which is also known as the Euler fluid equations. The fluid we are now trying to simulate is called an inviscid fluid. Even though we have removed the viscosity from the equations, it can still look like viscosity was part of the simulation. This is because of the numerical methods we use and the errors they introduce. An important reason why we prefer the hybrid Lagrangian/Eulerian method in this report is so to avoid a lot of the numerical dissipation shown in pure grid based solutions that tend to use a lot of linear interpolation.

At first, Equation 3 might look a bit complicated. To get a better understanding of the different terms, let us start with a simple example where the scalar quantity c is depending on where in space and time it is evaluated. c is said to be a function of spatial coordinates $\vec{x}$ and time t, $c(t, \vec{x})$. To find

out how much c is changing at coordinate $\vec{x}$, we take the temporal derivate of c.

$$\frac{d}{dt}c(t,\vec{x}) = \frac{\partial c}{\partial t} + \nabla c \cdot \frac{d\vec{x}}{dt} \tag{4}$$

Equation 4 looks a lot similar to the first two terms in Equation 3. Instead of having a scalar quanity c, we can take the temporal derivate of a vector $\vec{u}$ as can be seen in Equation 5.

$$\frac{d}{dt}\vec{u}(t,\vec{x}) = \frac{\partial \vec{u}}{\partial t} + \nabla u \cdot \vec{u} \tag{5}$$

Before we try to understand what Equation 3 really means, let us look at the following example:

$$\frac{\partial \vec{u}}{\partial t} + \nabla u \cdot \vec{u} = 0 \tag{6}$$

This tells us that there is no change in velocity, no matter where in time we are, which of course is false for most if not all fluids. Let us rearrange Equation 3 a little bit and move the pressure gradient part to the right hand side.

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = g - \frac{1}{\rho}\nabla p \tag{7}$$

What the Euler fluid equations really are telling us is that the change of velocity field of the fluid is equal to a combination of external forces and the negative pressure gradient. The external forces are in most cases static and the most of the interesting movement in a fluid comes from how much pressure varies in space.

The incompressibility part is explained in Equation 2 where it says that the velocity field of the fluid has to be divergence-free everywhere. This means that the volume can never change in the fluid. For example, let us imagine a case where we look at a subset region of the fluid, shaped as a box. The rest of the fluid is surrounding the box. Let us also say that the flow of the fluid is only one dimensional in this case and that it is flowing in the positive x-direction. The amount of fluid that is leaving our box region through the positive x-direction edge of the box has to be exactly the same as the amount entering through the negative x-direction edge (because of the one-dimensional flow). The divergence-free condition is going to play an important role later on when we solve for pressure at each step Δt.

2.2 MAC grid

When storing quantities on a grid, it is common to store the values at the center of each cell. This approach can be tempting to use since it makes implementations clean and simple. However, there are different ways to store the quantities and the values do not necessary have to be stored at the center of the cell. An example of this is the *Marker and Cell grid*, MAC grid, introduced by Harlow and Welch [10].

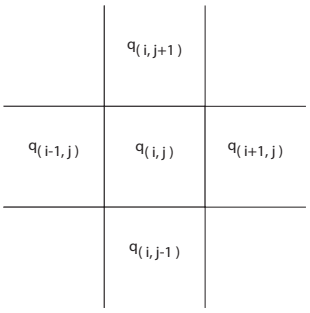


Figure 1: A grid with quantity q stored at the center of the cells.

Before we talk about the difference of a regular grid and a MAC grid, let us revisit derivative approximations. Figure 1 shows a simple 2D-dimensional grid with the quanity $q_{i,j}$ stored at the center of each cell. The cells are symmetric and we use the notation Δx for the cell width. To approximate $\frac{\partial q}{\partial x})_{i,j}$, we can either use first order forward difference

$$(\frac{\partial q}{\partial x})_{i,j} \approx \frac{q_{i+i,j} - q_{i,j}}{\Delta x} \tag{8}$$

accurate to $O(\Delta x)$ or use first order central difference

$$(\frac{\partial q}{\partial x})_{i,j} \approx \frac{q_{i+i,j} - q_{i-1,j}}{2\Delta x} \tag{9}$$

that has accuracy proportional to $O(\Delta x^2)$. Even though central differencing is more accurate, we can make it even more accurate without having to use complicated approximations. To solve the Navier-Stokes equations later on we are going to need to approximate partial derivatives of both the velocity field $\vec{u}$ and the pressure field p. As we will see in later sections of this report, the partial derivative of the pressure field $\frac{\partial p}{\partial x}$ is going to be evaluated to update the velocity field $\vec{u}$. In a similar way, we will have to evaluate $\nabla \cdot \vec{u}$ in the linear equations to solve for pressure. This motivates the use of a staggered grid, i.e a grid where different variables are stored at different locations in the grid.

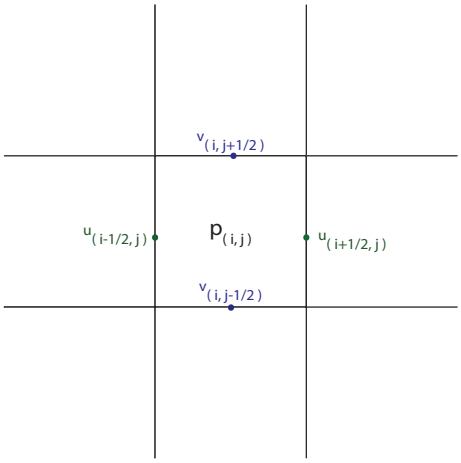


Figure 2: A MAC grid with pressure p stored at the center of the cell and velocties u and v stored at the edges.

In Figure 2 we see a two-dimensional MAC grid. A MAC grid stores the pressure p at the center of the cell and splits the vector field $\vec{u} = (u, v)$ into a component for each axis and store it on the edges of the cell, i.e. in this 2D example it splits (u, v) and stores u on the horizontal edges and v on the vertical ones. At first, this might be confusing but when we evaluate the derivatives it should make more sense. To evaluate the divergence of $\vec{u}$ at the center of a cell (i, j), we use central difference on the values stored at the edges.

$$\nabla \cdot \vec{u} = \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - u_{i,j-1/2}}{\Delta x} \tag{10}$$

Compared to Equation 9, we only divide by Δx instead of $2\Delta x$. In terms of implementation, the staggered grid does not have a larger memory footprint than a regular grid structure. In other ways, we get more accurate derivatives without slowing down the performance. To evaluate the derivative of the pressure p at a edge we perform central difference on neighboring pressure values stored at the center.

$$\frac{\partial \rho_{i-1/2,j}}{\partial x} = \frac{\rho_{i,j} - \rho_{i-1,j}}{\Delta x} \tag{11}$$

$$\frac{\partial \rho_{i,j-1/2}}{\partial y} = \frac{\rho_{i,j} - \rho_{i,j-1}}{\Delta x} \tag{12}$$

This report is frequently going to mention the use of barycentric coordinates. If vertices $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ span a rectangle, any point $\vec{v}_c$ inside the rectangle can then be represented as a linear combination of the vertices.

$$\vec{v_c} = \vec{b_0}\vec{v_0} + \vec{b_1}\vec{v_1} + \vec{b_2}\vec{v_2} + \vec{b_3}\vec{v_3} \tag{13}$$

where $\vec{b_i} = \{b_{i,x}, b_{i,y}\}$. The coefficients $\vec{b_i}$ are called the barycentric coordinates. They are used for bilinear interpolation, i.e. given a position $\vec{v_c}$ - how much do we need to interpolate from nearby vertices. They are also used for weighting where given a vertex $\vec{v_i}$, how much is a quantity at $\vec{v_c}$ affecting the quantity the vertex?

2.3 Level Set

Liquid fluids, unlike gas fluids, have a surface. Tracking a surface is a non-trivial problem to solve due to the complex shape and the evolving change of topology. It is important to get the surface tracking accurate as we later on we need to know if a grid cell is inside, outside or on the surface when solving for pressure. To track the surface, we will use a level set [11]. A level set is a grid approach to track a surface. At each cell in the grid, a scalar $\phi_{i,j}$ is stored. This absolute value of the scalar tells us the closest distance to the surface. To know if we are inside, outside or on the actual surface we use the following notation:

$$\phi = \begin{cases} \phi < 0 & \text{inside} \\ \phi > 0 & \text{outside} \\ \phi = 0 & \text{surface} \end{cases} \tag{14}$$

Figure 3 shows an example of a surface and a blue interior and a white exterior area. All the cells that have their centers outside of the blue area have positive values and the ones on the inside have negative values. Notice that if the surface is aligned on the center of a cell, ϕ is zero. The further away a cell is from the surface, the larger $|\phi|$ is.

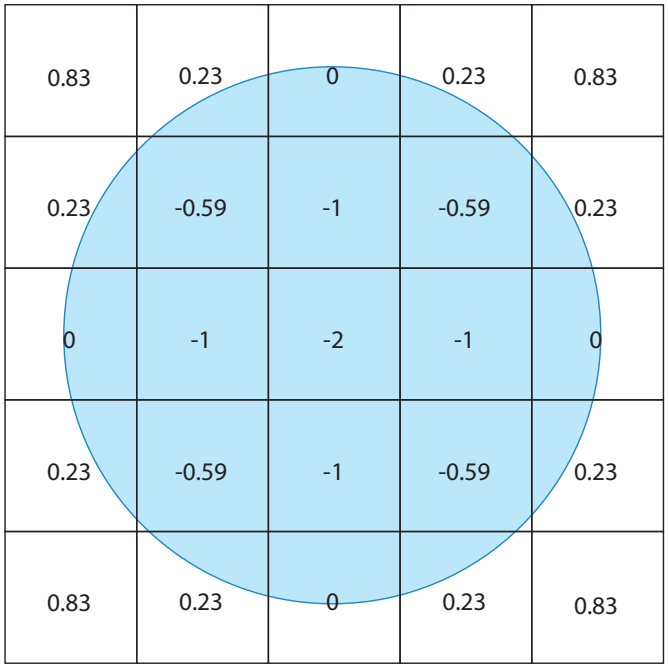


Figure 3: A numerical example of a level set representing a sphere (blue).

Another important feature of a level set is the gradient.

$$\nabla\phi = \hat{n} \tag{15}$$

where $\hat{n}$ is the unit vector describing the direction to the closest point of the surface. This means, given spatial coordinates $\vec{x}$, one can easily find the closest point $\vec{x}_s$ on the surface. If we evaluate the level set at $\vec{x}$, i.e. $\phi(\vec{x})$, we get the distance to the surface. If we walk this distance in the negative direction of the gradient, we end up at the surface.

$$\vec{x}_s = \vec{x} - \phi(\vec{x})\nabla\phi(\vec{x}) \tag{16}$$

For any level set operator, it is important that the length of the gradient is always of unit length. This is called the Eikonal equation.

$$|\nabla\phi| = 1 \tag{17}$$

2.4 Outline of the Algorithm

This section will briefly outline the important steps of the FLIP algorithm that are covered later on in the report. The key to the hybrid FLIP method is that we store both particles and a grid. Each particle stores a position in space, $\vec{x}_p$ and a velocity $\vec{u}_p$. The particles and the grid are going to comminucate and transfer properties to each other. The particles are integrated over time to update the positions. The particle velocities are later transferred to the grid and it is up to the grid to make the velocity field divergence-free and therefore conserving the volume of the water. The input to the algorithm is the initial geometry shape of the fluid and the collidable solid boundaries, i.e. walls or rigid objects that the water should collide against. We will refer to the solid boundaries as just solids in this report. We will use a level set to track the surface of the solids, ϕ_s. It is trivial to create the solid level set if the we only deal with solids at the grid boundaries. How to create more complex boundaries is not covered in this thesis. More information about creating level sets from arbitrary geometry can be found in [17].

1. Create particles from initial shape - *Ch 3.1*
2. Transfer particle velocities to grid - *Ch 3.2*
3. Apply external forces to grid
4. Mark cells fluid - *Ch 4.1*
5. Create level set - *Ch 4.1*
6. Reinitialize level set - *Ch 4.2*
7. Extrapolate velocities - *Ch 4.3*
8. Solve pressure equations- *Ch 5*
9. Transfer grid velocities to particles - *Ch 3.3*
10. Advect particles - *Ch 3.4*
11. Repeat step 2-10 until simulation done

The only item not covered later on in this report is *Apply external forces to grid*. This step is straightforward and can be done with a simple Euler step:

$$\vec{u}^{ext} = \vec{u} + \Delta t \cdot \vec{g} \tag{18}$$

3 Particles

3.1 Creating particles

We need to create the particles representing the volume of the water before we run the simulation. To do this we need a function that tells us if a point in space $\vec{x}$ is inside our outside of the initial shape of the water. We create c_p particles for a two dimensional cell that is completely inside of the water. The initial FLIP report suggests the use of $c_p = 4$ with particles randomly created inside each cell. Less particles tend to create gaps in the simulation and more particles create unnecessary noise.

Algorithm 1 Creating particles from an initial water shape

for $i = 0$ to N_x **do**
  for $j = 0$ to N_y **do**
    for $k = 0$ to c_p **do**
      p = (i,j)$\cdot\Delta x + \frac{random(-1,1)\cdot\Delta x}{2}$
      if p is inside water and $\phi_s(p) > 0$ **then**
        create particle at p
      end if
    end for
  end for
end for

In Algorithm 1, p is a two dimensional vector. *random(a,b)* is a function that returns a random value uniformly distributed between a and b. The $\phi_s > 0$ is a solid level set check to see if the position is outside of the solid. Figure 4 shows a basic example of an initial solid level set ϕ_s colored grey and an closed water volume in blue.

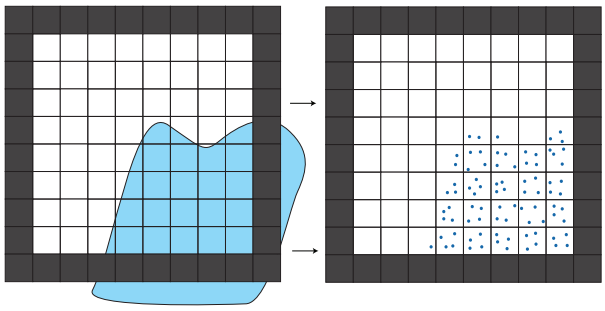


Figure 4: Creating the FLIP particles from a given shape.

3.2 Transfer particle velocities to grid

The first thing we need to do after advecting the particles is to transfer the particle velocities to our grid based velocity field. The velocities u and v are stored on different edges in the MAC grid and we need to be careful when we weight and transfer the particle velocities. The easy case would be when the particle position $\vec{x}_p$ is aligned exactly at the middle of the edge between two cells. Let us compare the difference between u and v.

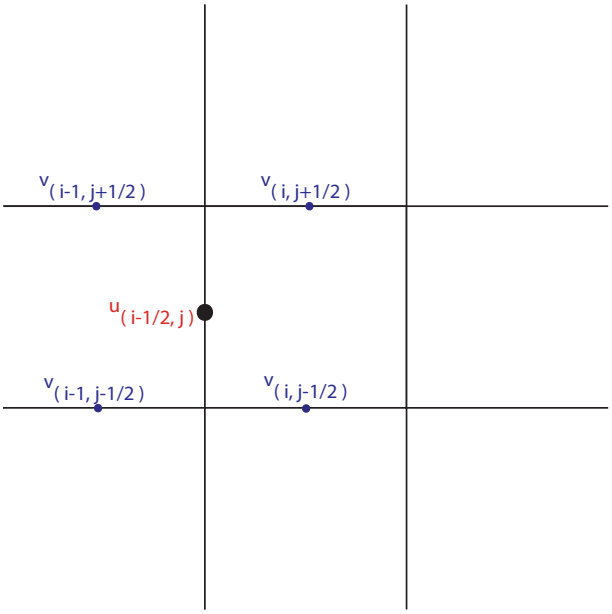


Figure 5: The particle position is aligned exactly at grid position $(i-1/2, j)$.

The particle position $\vec{x}_p$ in Figure 5 is exactly at grid position $(i-1/2, j)$. It is reasonable that the horizontal component of the particle velocity u_p only affects the edge at $u_{i-1/2,j}$. The vertical component v_p is not exactly aligned with any vertical edge. The position $\vec{x}_p$ is exactly in the middle of the rectangle that the vertical edges $v_{i-1,j-1/2}, v_{i,j-1/2}, v_{i-1,j+1/2}, v_{i,j+1/2}$ spans. In this case, it makes sense that the vertical component of the particle velocity v_p should affect the velocity field at all four vertical edges equally much since it it's in the middle of the rectangle.

What if there are multiple particles nearby? How do they all affect the velocities in the grid together? Figure 6 shows a case where multiple particles are close to a single velocity in the MAC grid. We need a way to weight the velocities so that particle velocities closer to an edge have more effect than the ones further away. We are going to use the barycentric coordinates

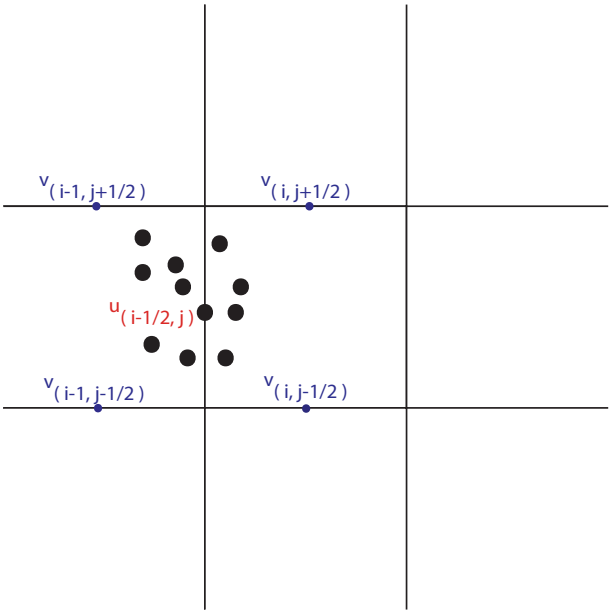


Figure 6: Multiple particles nearby a single horzintal edge at $(i - 1/2, j)$.

of a particles inside the rectangles spanned by nearby edges to decide how much of the particle velocity is going to affect the neighboring velocities in the MAC grid. The final velocity of an edge in the MAC grid is decided by

$$u_{i-1/2,j} = \frac{\sum_{i \in A} \omega_A(x_i) u_i}{\sum_{i \in A} \omega_A(x_i)} \tag{19}$$

$$v_{i,j-1/2} = \frac{\sum_{i \in B} \omega_B(x_i) v_i}{\sum_{i \in B} \omega_B(x_i)} \tag{20}$$

where A is the set of particles around a horizontal edge and $\omega_A(\vec{x})$ is the horizontal weighting function. B is then the set of particles around a vertical edge and $\omega_B(x_i)$ is the vertical weighting function. In the implementation, both $\omega_A(\vec{x})$ and $\omega_B(\vec{x})$ use the barycentric coordinates to decide the weights. Figure 7 shows the rectangular areas that decide the barycentric coordinates for a particle.

Figure 7 shows the difference between the sets A and B. Updating one edge at a time would require a sophisticated way of only accessing particles close to a specific edge. If not, the time complexity for gathering particles

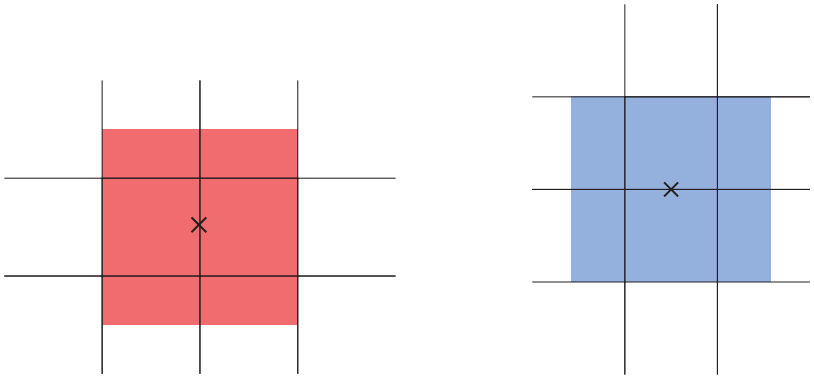


Figure 7: To the left, the velocity u marked as a cross at the edge is only updated from particles within the red area. To the right, the velocity v marked as a cross is only updated from the blue area.

in A and B would be $O(n)$ for each edge where n is the total number of particles in the simulation. Instead, we will use a technique called splatting. In splatting approaches, you iterate over all the particles instead of all the grid cells. For each particle, we find out which edges it could affect. This is fast since each particle has a position and A and B have a fixed maximum size. The splatting method is divided into two parts. Part one can be seen in Algorithm 2. Notice that there is a temporary *sum* MAC grid for storing the denominator part of Equation 19.

The barycentric functions in Algorithm 2 returns two normalized coordinates from 0 to 1 where bottom left of the region gives back $(0, 0)$, bottom right $(1, 0)$, top left $(0, 1)$ and top right $(1, 1)$. Anything in between is linearly interpolated.

The second step of the splatting process is to normalize all the velocities. In Algorithm 2 we only applied the numerator of Equation 19. Once all velocities are splatted we also have the sum of all weights. If we divide the current edge velocity with the sum stored at each edge, Equation 19 is true and the transfer from particles to grid is done.

Algorithm 2 Step one in splatting particle velocities to grid velocities

```
grid = empty mac grid
sum = empty mac grid
for all particles p do
  tx,ty = barycentricA(p.x)
  for neighboring horizontal edges e do
    weight = omega(tx,ty, p.x)
    grid.u[e.idx] += weight * p.u
    sum.u[e.idx] += weight
  end for
  tx,ty = barycentricB(p.x)
  for neighboring vertical edges e do
    weight = lambda(tx,ty, p.x)
    grid.v[e.idx] += weight * p.v
    sum.v[e.idx] += weight
  end for
end for
```

Algorithm 3 Step two in splatting particle velocities to grid velocities

```
for i = 0 to N_x do
  for j = 0 to N_y do
    idx = .... get index for (i-1/2,j)
    grid.u[idx] /= sum.u[idx]
    idx = ... get index for (i,j-1/2)
    grid.v[idx] /= sum.v[idx]
  end for
end for
```

3.3 Transfer grid velocities to particles

As we could see in the previous section, going from particle velocities to grid can be a bit tricky when working with a MAC grid. Luckily, going from grid velocities to particle velocities is a lot easier. For every particle, we will use bilinear interpolation to get the horizontal and vertical velocities.

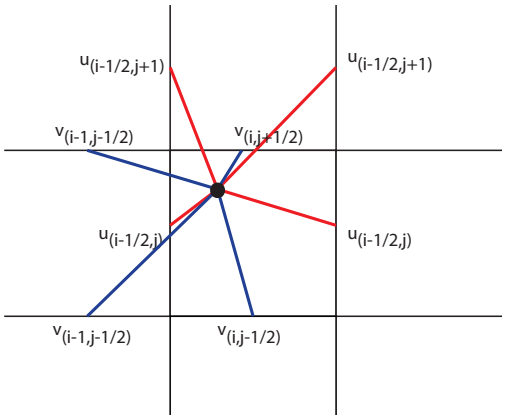


Figure 8: The particle velocity $\vec{u}_p$ is bilineary interpolated between neighboring velocities in the MAC grid.

The FLIP part of our hybrid particle-grid approach is how we update the particle velocities. If we only update the particles with the velocities that the pressure solving step gives us, it is called *Particle in Cell*, PIC, which is an older approach. Zhu and Bridson [2] are in their FLIP paper, before solving the pressure equations, saving the old divergence-free velocity field and then updating the particles with the change of velocity instead of only the new velocity.

$$\delta\vec{u} = \vec{u}^{n+1} - \vec{u}^{n} \tag{21}$$

FLIP alone can cause a lot of noise because of the large particle count. The FLIP paper recommends linear interpolation between FLIP and PIC for best result. The formula for updated particle velocities can be expressed as:

$$\vec{u}_p = \alpha \cdot bilerp(\vec{u}^{n+1}, \vec{x}_p) + (1-\alpha) \cdot (\vec{u}_p^{old} + bilerp(\delta\vec{u}, \vec{x}_p)) \tag{22}$$

where α is a number between 0 and 1. If one, it is only PIC and zero is pure FLIP. Values closer to one gives a more stable look but to the cost of more numerical dissipation which cancels out a lot of the interesting high frequency motion. The $bilerp(\vec{u}, \vec{x})$ function bilineary interpolates a velocity in the MAC grid at position $\vec{x}$. An example of this can be seen in Figure 8 where the black particle is at position $\vec{x}$.

3.4 Advecting particles

The most straight forward approach to update particle positions $\vec{x}_p^{n+1}$ is to use simple forward Euler time integration on the velocities given from Equation 21.

$$\vec{x}_p^{n+1} = \vec{x}_p^n + \Delta t \vec{u}_p \tag{23}$$

We can improve accuracy by using a different time integration scheme. First we need Equation 21 to be a function that uses an arbitrary position $\vec{x}$ as input and not the position from each particle $\vec{x}_p$. We call this function $vel(\vec{x})$.

$$vel(\vec{x}) = \alpha \cdot bilerp(\vec{u}^{n+1}, \vec{x}) + (1 - \alpha) \cdot (\vec{u}_p^n + bilerp(\delta\vec{u}, \vec{x})) \tag{24}$$

We are going to use a second order Runge-Kutta method where we evaluate a mid point and use the mid point to evaluate the velocity of the particle rather than the position of the particle.

$$\vec{x}_p^{n+1/2} = \vec{x}_p^n + \frac{\Delta t}{2} vel(\vec{x}_p) \tag{25}$$

Note that we only use half of the timestep Δt in Equation 24 to evaluate the in-between position $\vec{x}_p^{n+1}$. The final position of a particle in each step can then be set to

$$\vec{x}_p^{n+1} = \vec{x}_p^n + \Delta t \cdot vel(\vec{x}_p^{n+1/2}) \tag{26}$$

After advecting the particles, we still need to run a correction step in case some of the particles get stuck in the solid. The boundary condition of the pressure equations (covered later in this report) are supposed to prevent this from happening but there are still going to be cases where particles intersect the solid due to numerical errors in our single precision simulation. In Chapter 2.4, we introduced the solid level set ϕ_s. We move the particle out in the normal direction of the solid if a particle is found inside the level set ϕ_s.

$$\vec{x}_p^{correction} = \vec{x}_p + \beta |\phi_s| \nabla \phi_s \tag{27}$$

with $\beta > 1$. If β would be exactly 1 then we would only move the particle to surface of the solid. In the implementation, $\beta = 1.5$ was used.

4 Surface tracking

4.1 Creating the Level Set

Before we create the level set ϕ that represent the surface of our water simulation, we need to mark the cells we consider being fluid. The easiest approach is to go from particle position $\vec{x}_p$ to grid coordinates (i, j) and tag the cell with those coordinates as fluid.

Algorithm 4 Marking cells as fluid

```
marker = empty grid (all values 0)
for all particles p do
  i,j = int(p.x / Δx)
  marker(i,j) = 1
end for
```

In Algorithm 4, a cell will most likely contain several particles at once which means that we will write the value 1 to that cell multiple times. Although unnecessary, this is a very fast operation compared to all the other steps in the level set creation and the overhead is not noticeable. Figure 9 shows an example of the marker grid after we run Algorithm 4 on an input of particles. Notice that cells with only one particle is still counted as a fluid cell.

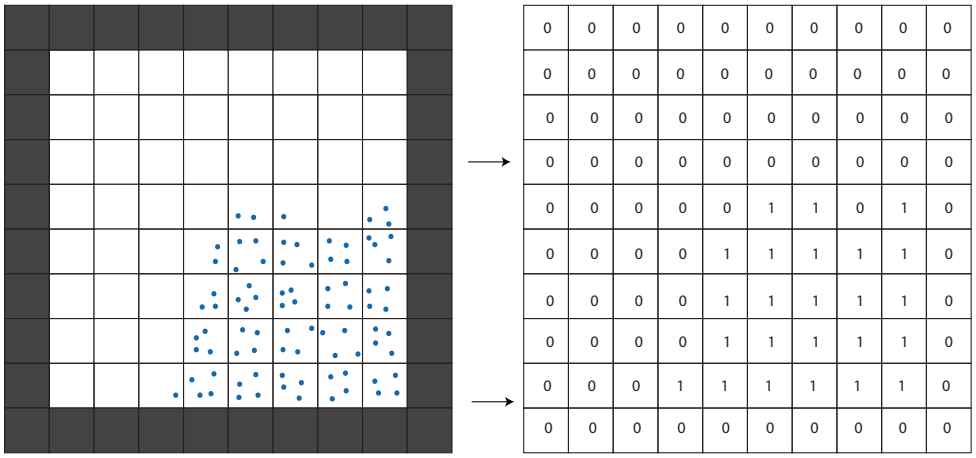


Figure 9: Cells with particles inside them as are marked as fluid cells.

Then next step in the algorithm is to initialize the level set. This step will not create a valid level set that satisfies Equation 16. To start with, we set $\phi_{i,j}$ to $-\frac{\Delta x}{2}$ as if the surface could be at the edge of the cell. To cells not marked

as fluid we set the value to $D \cdot \Delta x$ where it is important that $D >> \Delta x$. Typically the initializing step is run with $D = 1000$.

Algorithm 5 Initalizing the Level Set

```
marker = ... (marker grid from Algorithm 4)
inside = -0.5 * Δx
outside = D * Δx
for i = 0 to N_x do
  for j = 0 to N_y do
    if marker(i,j) is 1 then
      phi(i,j) = inside
    else
      phi(i,j) = outside
    end if
  end for
end for
```

After the initializing step, the level set is far from continuous. To make it continuous and satisfy $|\nabla\phi| = 1$, we are going to iteratively solve the Eikonal equation. This step is sometimes called the reinitializing step in level set literature.

4.2 Reinitializing the Level Set

The goal of the reinitializing step is to make sure that the gradient of the level set is always a unit vector, i.e the length is 1. To do this we will use a method called fast sweeping, which sweeps and propagates the solution of an iterative approach in different directions. Before we go into detail about the fast sweeping algorithm itself, let us write down and evaluate tha gradient of ϕ and the Eikonal equation in two dimensions

$$\nabla\phi = (\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}) \tag{28}$$

$$|\nabla\phi| = \sqrt{(\frac{\partial\phi}{\partial x})^2 + (\frac{\partial\phi}{\partial y})^2} = 1 \tag{29}$$

We can now turn Equation 28 into an discrete version and therefore making it possible to solve it iteratively. Let us approximate the partial derivatives with backward differences

$$\begin{aligned} \frac{\partial\phi}{\partial x} &\approx \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} \\ \frac{\partial\phi}{\partial y} &\approx \frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta x} \end{aligned} \tag{30}$$

If we square both the left and right hand side of Equation 28 and use the approximations of the partial derivatives, we get

$$\begin{aligned} (\frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x})^2 + (\frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta x})^2 &= 1 \\ ({\phi_{i,j}}^2 - 2\phi_{i,j}\phi_{i-1,j} + {\phi_{i-1,j}}^2) + ({\phi_{i,j}}^2 - 2\phi_{i,j}\phi_{i,j-1} + {\phi_{i,j-1}}^2) &= \Delta x^2 \\ 2{\phi_{i,j}}^2 - 2\phi_{i,j}(\phi_{i-1,j} + \phi_{i,j-1}) + {\phi_{i-1,j}}^2 + {\phi_{i,j-1}}^2 - \Delta x^2 &= 0 \\ {\phi_{i,j}}^2 - \phi_{i,j}(\phi_{i-1,j} + \phi_{i,j-1}) + \frac{{\phi_{i-1,j}}^2 + {\phi_{i,j-1}}^2 - \Delta x^2}{2} &= 0 \end{aligned} \tag{31}$$

To simplify things, let us store the non $\phi_{i,j}$ parts in temporary variables

$$\begin{aligned} b &= \phi_{i-1,j} + \phi_{i,j-1} \\ c &= \frac{{\phi_{i-1,j}}^2 + {\phi_{i,j-1}}^2 - \Delta x^2}{2} \end{aligned} \tag{32}$$

Equation 30 can then be put on the form

$$\phi_{i,j}{}^2 - \phi_{i,j}b + c = 0 \tag{33}$$

which is a standard quadratic equation. Rearranging the terms we get

$$\begin{aligned} \phi_{i,j}{}^2 - \phi_{i,j}b + c &= 0 \\ (\phi_{i,j} - \frac{b}{2})^2 &= b^2 - c \\ \phi_{i,j} &= \frac{b}{2} \pm \sqrt{b^2 - c} \end{aligned} \tag{34}$$

If $b^2 - c < 0$, we do not update $\phi_{i,j}^{n+1}$. If $b^2 - c > 0$, Equation 33 has two real solutions $\phi_{i,j}{}^a$ and $\phi_{i,j}{}^b$. We pick the smaller value of the two as our solution.

$$\phi_{i,j}{}^{new} = min(\phi_{i,j}{}^a, \phi_{i,j}{}^b) \tag{35}$$

Another condition we use is that new solution to $\phi_{i,j}{}^{new}$ has to be smaller than the previous value $\phi_{i,j}{}^n$. This is why it was important to pick $D >> \Delta x$. It means that we are only propagating the level set from the surface and inwards/outwards and not from the center of the fluid towards the surface.

$$\phi_{i,j}{}^{n+1} = \begin{cases} \phi_{i,j}{}^{new} & \text{if } \phi_{i,j}{}^{new} < \phi_{i,j}{}^n \\ \phi_{i,j}{}^n & \text{if } \phi_{i,j}{}^{new} > \phi_{i,j}{}^n \end{cases} \tag{36}$$

Now that we have an iterative way of solving the Eikonal equation, let us talk more about the fast sweeping method. Fast sweeping for solving the Eikonal equation was introduced by Zhao[4]. The fundamentals of the method are that you sweep your domain in every possible direction which tells you in which order to update the cells. In a two dimensional case, there are only four different directions:

- left to right, bottom to top
- left to right, top to bottom
- right to left, bottom to top
- right to left, top to bottom

Depending on the sweeping order, the approximation of the partial derivatives in Equation 29 is different. The goal with fast sweeping is to use updated data from the same sweep and therefore have faster convergence. For example, if we are sweeping from left to right, it makes sense to approximate the

horizontal partial derivative with $\phi_{i,j} - \phi_{i-1,j}$ since $\phi_{i-1,j}$ was just updated. However, if we are sweeping from right to left, then it is better to approximate the horizontal partial derivative with $\phi_{i,j} - \phi_{i+1,j}$. We need to take this into account when we are solving Equation 33.

$$\frac{\partial \phi}{\partial x} \approx \begin{cases} \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} & \text{if horizontal sweeping is left to right} \\ \frac{\phi_{i,j} - \phi_{i+1,j}}{\Delta x} & \text{if horizontal sweeping is right to left} \end{cases} \tag{37}$$

$$\frac{\partial \phi}{\partial y} \approx \begin{cases} \frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta x} & \text{if vertical sweeping is bottom to top} \\ \frac{\phi_{i,j} - \phi_{i,j+1}}{\Delta x} & \text{if vertical sweeping is top to bottom} \end{cases} \tag{38}$$

Figure 10 shows an example of the update order when sweeping from left to right and bottom to right.

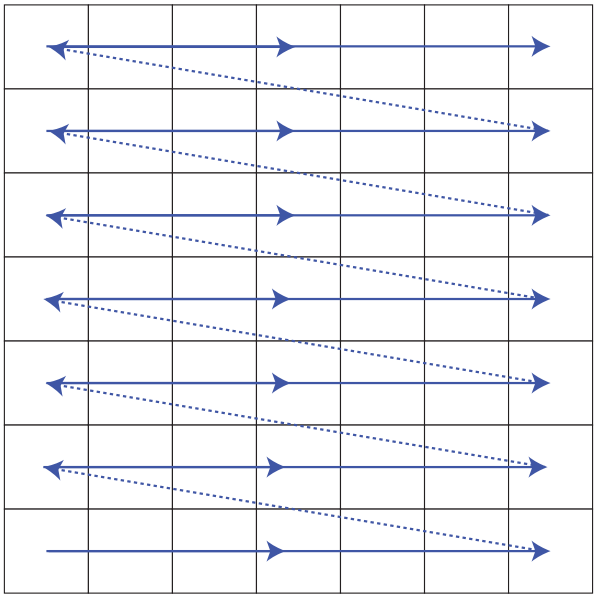


Figure 10: Left to right, bottom to top sweeping order. In an iterative solver, the cells are updated in the order of the blue arrows.

4.3 Extrapolate velocities

Sometimes it happens that an edge on the grid is close to the surface but no particle was close enough to update the velocity of the edge. We have to be able to guarantee that we have valid velocities on the grid where the level set tells us an edge inside or close to the surface of the fluid. To ensure this, we will extrapolate the velocities from the surface and outwards with the help of the level set. Remember from section 2.3 that the gradient of the level set is the same as the normal of the surface. If we extrapolate the velocities in the direction of the normal, the derivative of the velocities along the normal should be zero. We can use this condition to create an iterative solution to the unknown velocities outside of the surface. The following equations say that the horizontal velocity u should be constant along the normal of the surface.

$$\begin{aligned} \nabla\phi \cdot \nabla u &= 0 \\ \frac{\partial\phi}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial\phi}{\partial y}\frac{\partial u}{\partial y} &= 0 \end{aligned} \tag{39}$$

To find an expression that we can solve iteratively we need to approximate the partial derivatives similar to what we did in previous section. We can then use the fast sweeping method once more. Notice that Equation 38 is only for the horizontal velocities u and one has to solve vertical velocities v individually. It is important that the fast sweeping algorithm only updates velocities on edges that did not receive any velocities from the particles because we do not want the velocity field to be constant in the negative direction of the normal.

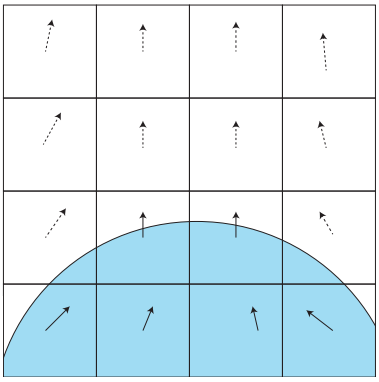


Figure 11: The velocities of the cells inside the fluid (marked as solid arrows) are extrapolated in the surface normal direction (marked as dashed arrows).

5 Solving for pressure

To guarantee that the velocity field is divergence-free we need to solve the pressure equations. This step is often called the projection step. We are going to use an approach introduced by Alexandre Chorin introduced [12]. The key is to split the Euler fluid equations into two steps and solve them sequentially. In the first step we solve for the intermediate velocity field $\vec{u}^*$. This vector field is not divergence-free but we use it as an input to the second step of the splitting. We will set up the pressure equations later on in such way that removing the pressure gradient ∇p from the intermediate $\vec{u}^*$ will make $\vec{u}^{n+1}$ divergence-free.

$$\vec{u}^{n+1} = \vec{u}^* - \frac{\Delta t}{\rho}\nabla p \tag{40}$$

If we rearrange Equation 39, we get a substitution for the pressure term in the Euler fluid equations described in Equation 3.

$$\frac{\vec{u}^{n+1} - \vec{u}^*}{\Delta t} = -\frac{1}{\rho}\nabla p \tag{41}$$

If we approximate $\frac{\partial \vec{u}}{\partial t}$ with $\frac{\vec{u}^{n+1}-\vec{u}^n}{\Delta t}$, we can rewrite Equation 3 to

$$\begin{aligned} \frac{\partial \vec{u}}{\partial t} &= -\vec{u}\cdot\nabla\vec{u} - \frac{1}{\rho}\nabla p + \vec{f} \\ \frac{\vec{u}^{n+1} - \vec{u}^n}{\Delta t} &= -\vec{u}\cdot\nabla\vec{u} + \frac{\vec{u}^{n+1} - \vec{u}^*}{\Delta t} + \vec{f} \\ \vec{u}^* &= \vec{u}^n - \Delta t(\vec{u}\cdot\nabla\vec{u}) + \Delta t\vec{f} \end{aligned} \tag{42}$$

In our hybrid FLIP simulation, getting the intermediate velocity field $\vec{u}^*$ each step is easy where the Lagrangian particle update gives the use the advection terms. Applying the integrated external forces $\vec{g}$ over timestep Δt is trivial. The challenging part of the pressure solving step is to set up the pressure equations that enforces an incompressible fluid.

5.1 Setting up the Pressure Equations

Let us look at the discrete implementation of Equation 39. The use of a MAC grid comes in handy once again since it makes the use of central difference approximations easy and more robust. For each dimension, we perform a pressure update to the velocity field $\vec{u}^{n+1}$ the following way:

$$\begin{aligned} u^{n+1}_{i+1/2,j} &= {u^*}_{i+1/2,j} - \frac{\Delta t}{\rho}\frac{p_{i+1,j} - p_{i,j}}{\Delta x} \\ u^{n+1}_{i-1/2,j} &= {u^*}_{i-1/2,j} - \frac{\Delta t}{\rho}\frac{p_{i,j} - p_{i-1,j}}{\Delta x} \\ v^{n+1}_{i,j+1/2} &= {v^*}_{i,j+1/2} - \frac{\Delta t}{\rho}\frac{p_{i,j+1} - p_{i,j}}{\Delta x} \\ v^{n+1}_{i,j-1/2} &= {v^*}_{i,j-1/2} - \frac{\Delta t}{\rho}\frac{p_{i,j} - p_{i,j-1/2}}{\Delta x} \end{aligned} \tag{43}$$

The most important condition is the incompressible one. We have to guarantee that for every cell in the grid, the next velocity field $\vec{u}^{n+1}$is divergence-free. If we evaluate the divergence-free condition for a cell in our MAC grid we get the following:

$$\frac{u^{n+1}_{i+1/2,j} - u^{n+1}_{i-1/2,j}}{\Delta x} + \frac{v^{n+1}_{i,j+1/2} - v^{n+1}_{i,j-1/2}}{\Delta x} = 0 \tag{44}$$

If we replace the velocities for next step $(n+1)$ with the right hand sides of Equation 42, we get

$$\begin{aligned} \frac{1}{\Delta x}[{u^*}_{i+1/2,j} - \frac{\Delta t}{\rho}\frac{p_{i+1,j} - p_{i,j}}{\Delta x} - {u^*}_{i-1/2,j} + \frac{\Delta t}{\rho}\frac{p_{i,j} - p_{i-1,j}}{\Delta x} \\ + {v^*}_{i,j+1/2} - \frac{\Delta t}{\rho}\frac{p_{i,j+1} - p_{i,j}}{\Delta x} - {v^*}_{i,j-1/2} + \frac{\Delta t}{\rho}\frac{p_{i,j} - p_{i,j-1/2}}{\Delta x}] = 0 \end{aligned} \tag{45}$$

If we rearrange the terms in Equation 44, we get an expression that is more intuitive to put in a linear system of equations.

$$\begin{aligned} 4p_{i,j} - p_{i-1,j} - p_{i+1,j} - p_{i,j-1} - p_{i,j+1} = \\ - \frac{\rho \Delta x}{\Delta t}({u^*}_{i+1/2,j} - {u^*}_{i-1/2,j} + {v^*}_{i,j+1/2} - {v^*}_{i,j-1/2}) \end{aligned} \tag{46}$$

This is the finite scheme of the Poission equation we solve for pressure, with the divergence of the velocity field as the right hand side.

5.2 Boundary conditions

There are two different kinds of boundary conditions we must take into account before we build our linear system. The first one is on the boundary between air and the fluid, also called as the free surface boundary condition. Air is a lot lighter than water (the density of water is approximately 700 larger than air) and we make the simplification in our model that air can be represented with constant atmospheric pressure (in reality air is also a fluid). Equation 3 tells us that only the derivative of pressure matters when updating our velocity field which means it does not matter which constant we use for air since the derivative is going to be zero. For convenience, we choose pressure to be zero for every air cell. When enforcing a constant at the boundary, it is called a Dirichlet boundary condition. In Equation 45, if a neighboring cell is air, we force the pressure of that cell to be zero. For example, when setting up the pressure equations for cell (i, j), let us say that cell $(i-1, j)$ is an air cell. The pressure equation for cell (i, j) is then changed to

$$\begin{aligned} 4p_{i,j} - 0 - p_{i+1,j} - p_{i,j-1} - p_{i,j+1} = \\ - \frac{\rho \Delta x}{\Delta t}(u^*{}_{i+1/2,j} - u^*{}_{i-1/2,j} + v^*{}_{i,j+1/2} - v^*{}_{i,j-1/2}) \end{aligned} \tag{47}$$

The more difficult boundary condition is the one between the fluid and the solid. In terms of our velocity field, it is important that no fluid is flowing into the solid. This means that the normal component of the velocity has to be zero

$$\vec{u} \cdot \hat{n} = 0 \tag{48}$$

In our implementation we are going to enforce the velocity of every edge next to a solid cell to be zero after updating the pressure. The confusing part is to understand how solid boundaries change the pressure equation. To find an expression for pressure in neighboring solid cells, let us start with an example where cell (i, j) is a fluid and cell $(i + 1, j)$. We want to guarantee that $u^{n+1}_{i+1/2,j} = 0$ after the pressure update, which means we can rewrite Equation 42 to

$$0 = u^*{}_{i+1/2,j} - \frac{\Delta t}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{49}$$

If we rearrange the terms we get an expression for the pressure inside the solid, $p_{i+1,j}$.

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t} u^*{}_{i+1/2,j} \tag{50}$$

In this example, where cell (i, j) has one solid neighboring cell at $(i + 1, j)$ and three neighboring fluid cells, we substitute $p_{i+1,j}$ in Equation 45 with the right hand side of Equation 49 to get

$$\begin{aligned} 4p_{i,j} - p_{i-1,j} - (p_{i,j} + \frac{\rho\Delta x}{\Delta t} u^*{}_{i+1/2,j}) - p_{i,j-1} - p_{i,j+1} = \\ - \frac{\rho\Delta x}{\Delta t}(u^*{}_{i+1/2,j} - u^*{}_{i-1/2,j} + v^*{}_{i,j+1/2} - v^*{}_{i,j-1/2}) \end{aligned} \tag{51}$$

which can be simplified to

$$3p_{i,j} - p_{i-1,j} - p_{i,j-1} - p_{i,j+1} = -\frac{\rho\Delta x}{\Delta t}(-u^*{}_{i-1/2,j} + v^*{}_{i,j+1/2} - v^*{}_{i,j-1/2}) \tag{52}$$

We now have everything we need to know to build a linear system of the pressure equations $A\vec{p} = \vec{b}$.

Algorithm 6 Building right hand side b

```
b = empty grid
scale = -ρΔx/Δt
for i = 0 to N_x do
  for j = 0 to N_y do
    if cell (i − 1, j) is not solid then
      b(i,j) += u*_{i−1/2,j}
    end if
    if cell (i + 1, j) is not solid then
      b(i,j) += u*_{i+1/2,j}
    end if
    if cell (i, j − 1) is not solid then
      b(i,j) += v*_{i,j−1/2}
    end if
    if cell (i, j + 1) is not solid then
      b(i,j) += v*_{i,j+1/2}
    end if
    b(i,j) *= scale
  end for
end for
```

In $A\vec{p} = \vec{b}$, A is a sparse matrix. Equation 45 is in fact a Laplacian equation in disguise. This means, at maximum, we only need to store 7 elements per row in the sparse matrix A. In Algorithm 7, we will use the notation of $A_{i,j}{}^{\{center,left,right,bottom,top\}}$ where subscript (i, j) tells us which row in A it corresponds to and superscript tells us which column on the row.

Algorithm 7 Building Sparse Matrix A

```
A = empty sparse matrix
for i = 0 to N_x do
  for j = 0 to N_y do
    if cell (i,j) is fluid then
      temp = 4
      if cell (i+1,j) is solid then
        temp -= 1
      else if cell (i+1,j) is fluid then
        A_{i,j}^{right} = -1
      end if
      if cell (i-1,j) is solid then
        temp -= 1
      else if cell (i-1,j) is fluid then
        A_{i,j}^{left} = -1
      end if
      if cell (i,j+1) is solid then
        temp -= 1
      else if cell (i,j+1) is fluid then
        A_{i,j}^{top} = -1
      end if
      if cell (i,j-1) is solid then
        temp -= 1
      else if cell (i,j-1) is fluid then
        A_{i,j}^{bottom} = -1
      end if
      A_{i,j}^{center} = temp
    end if
  end for
end for
```

5.3 Red Black Gauss-Seidel iterations

We are going to use an iterative solving method called Red-Black Gauss-Seidel to solve the linear system $A\vec{p} = \vec{b}$. A is a Laplacian matrix and therefore each equation in the linear system has the following form

$$A_{i,j}^{center} p_{i,j} + A_{i,j}^{left} p_{i-1,j} + A_{i,j}^{right} p_{i+1,j} + A_{i,j}^{bottom} p_{i,j-1} + A_{i,j}^{top} p_{i,j+1} = b_{i,j} \quad (53)$$

If we rearrange Equation 52 and use the notion p^n for current pressure value and p^{n+1} for the next updated value we get

$$p_{i,j}^{n+1} = -\frac{A_{i,j}^{left} p_{i-1,j}^{n} + A_{i,j}^{right} p_{i+1,j}^{n} + A_{i,j}^{bottom} p_{i,j-1}^{n} + A_{i,j}^{top} p_{i,j+1}^{n}}{A_{i,j}^{center}} + b_{i,j} \quad (54)$$

If we would iterate over all cells in our grid in each iteration step and use Equation 53, it would be called Jacobi iterations. Instead, we are going to use a Gauss-Seidel approach that converges faster than Jacobi iterations. The key is in which order we update the pressure. In Red-Black Gauss-Seidel we only update the pressure in a cell every other iteration step. If a cell (i, j) is updated in an iteration step n, the neighbors of that cell will not be updated. In iteration step $(n + 1)$, the cell (i, j) is not updated but instead all the neighbors are. Figure 12 shows how all the cells in a two dimensional grid are being divided to either a red or black set. On even iteration steps, we update the red cells and on uneven steps we update the black cells.

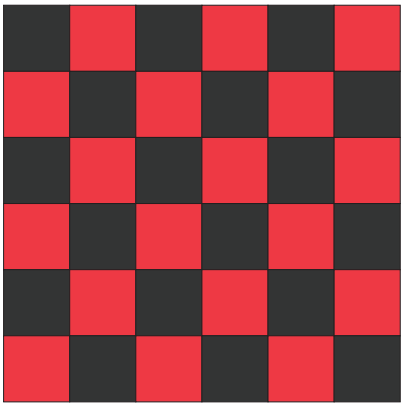


Figure 12: In Red Black Gauss-Seidel iterations, the grid domain is divided in to a red and a black region.

5.4 Restriction and Prolongation operations

Until this moment we have not talked about the grid size N_x and N_y. We only allow the grid size to be a power of two to make things as simply as possible.

$$2^M = max(N_x, N_y) \tag{55}$$

Before we break down the multigrid algorithm itself, we need a method to go from a matrix Q^M with resolution N_x, N_y to a lower resolution matrix Q^{M-1} with dimensions half the size, i.e. $N_x/2, N_y/2$. In general, we need an operation that uses a matrix Q^{M-m} with dimensions $\frac{N_x}{2^m}$ as an input and produces a down-scaled matrix Q^{M-m-1}. We call this operation restriction. We will use bilinear interpolation for downscaling a matrix.

$$Q_{i,j}^{M-n-1} = \frac{1}{4}(Q_{2i,2j}^{M-n} + Q_{2i+1,2j}^{M-n} + Q_{2i+1,2j+1}^{M-n} + Q_{2i,2j+1}^{M-n}) \tag{56}$$

Figure 13 gives a numerical example of the restriction operation seen in Equation 55

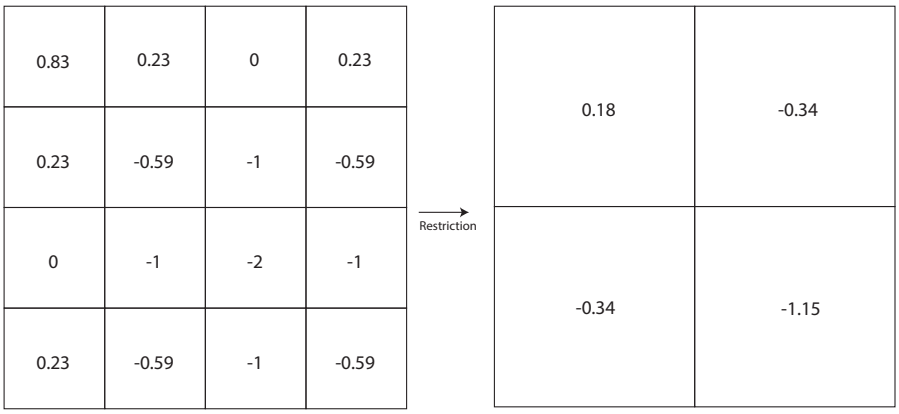


Figure 13: A numerical example of the restriction operation.

To keep things simple, we also use bilinear interpolation in the prolongation operator where we do the opposite from restriction, i.e. go from matrix Q^{M-m} to matrix Q^{M-m+1}. Although using the same interpolation scheme, prolongation is a bit trickier to get right since the indices are not aligned up as perfect as in the restriction operator. Figure 14 demonstrates why the indices are aligned slightly awkward.
The center of a cell in grid Q^{M-n} gives us the barycentric coordinates between nearby values in Q^{M-n-1} as can be seen in Figure 14. This gives us the following scheme for the prolongation operator:

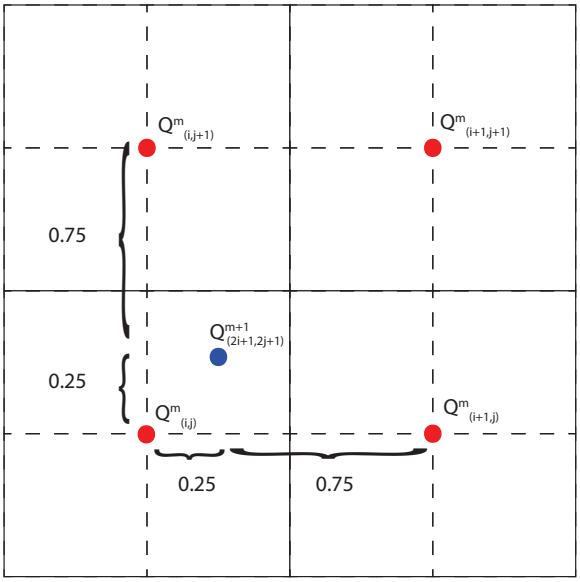


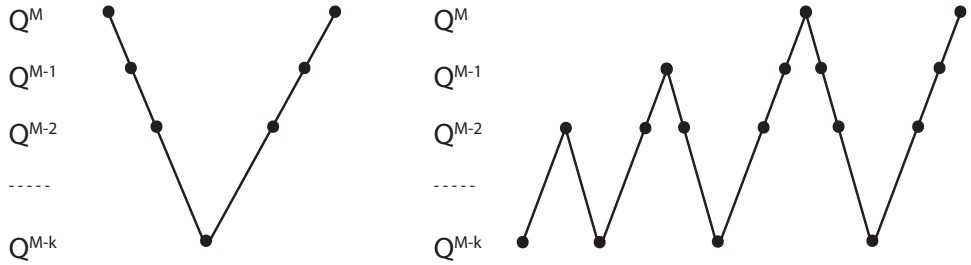
Figure 14: Barycentric coordinates are used to determine bilinear interpolation in the prolongation operator.

$$
\begin{aligned}
Q^{M-n}_{2i,2j} &= \frac{1}{16}Q^{M-n-1}_{i-1,j-1} + \frac{3}{16}Q^{M-n-1}_{i-1,j} + \frac{3}{16}Q^{M-n-1}_{i,j-1} + \frac{9}{16}Q^{M-n-1}_{i,j} \\
Q^{M-n}_{2i+1,2j} &= \frac{1}{16}Q^{M-n-1}_{i+1,j-1} + \frac{3}{16}Q^{M-n-1}_{i,j-1} + \frac{3}{16}Q^{M-n-1}_{i+1,j} + \frac{9}{16}Q^{M-n-1}_{i,j} \\
Q^{M-n}_{2i,2j+1} &= \frac{1}{16}Q^{M-n-1}_{i-1,j+1} + \frac{3}{16}Q^{M-n-1}_{i-1,j} + \frac{3}{16}Q^{M-n-1}_{i,j+1} + \frac{9}{16}Q^{M-n-1}_{i,j} \\
Q^{M-n}_{2i+1,2j+1} &= \frac{1}{16}Q^{M-n-1}_{i+1,j+1} + \frac{3}{16}Q^{M-n-1}_{i+1,j} + \frac{3}{16}Q^{M-n-1}_{i,j+1} + \frac{9}{16}Q^{M-n-1}_{i,j}
\end{aligned}
\tag{57}
$$

Equation 56 only works if the updated cells are not on the boundary and has four neighboring cells in the lower dimension grid. If the cell is on the boundary, we use simple linear interpolation between the two cells in Q^{M-n-1} with weights 0.75 and 0.25. There is one last special case. At the four corners, we use the same value as the corners of the lower dimension grid since there is nothing nearby to interpolate from.

5.5 The Multigrid method

One problem with iterative solvers of linear systems is that solutions propagate slowly. In Equation 53, only neighboring cells are part of this expression and therefore information only moves one cell per iteration step. This means that the larger the grid sizes N_x, N_y are, the longer it takes to reach convergence. We are not guaranteed that our velocity field is divergence-free if the solution to $Ap = b$ does not convergence. However, choosing a low resolution grid that converges faster introduces other non-wanted artifacts. An approach with high accuracy and fast convergence is desirable. The multigrid method tries to satisfy this where the idea is to solve the linear system in different resolutions and then use restriction and prolongations operations to transfer the answer from one resolution to another. There are different approaches to reach convergence and in this report we will focus on *Full Cycle* and *V-Cycle*. FIgure 15 demonstrates the difference between the two. Note that the Full Cycle uses incremental V-Cycles, something we can take advantage of when implementing the Full Cycle.



(a) V cycle (b) Full cycle

Figure 15: The two different multigrid schemes presented in this report

Using restriction and prolongation operations on the pressure grid in the V-cycles act as a low pass filter and this an unwanted effect. Instead of moving the pressure solution from one resolution to another, we will use restriction on the residual $r^{M-n} = b^{M-n} - A^{M-n}p^{M-n}$ and prolongation on the pressure p. Instead of solving $Ap = b$ in each resolution, we solve $Ap = r$. To update the pressure of level $M - n$ we use the linear combination of p^{M-n} and $prolong(p^{M-n-1})$.

$$p_{n+1}^{M-n} = p_n^{M-n} + prolong(p^{M-n-1}) \tag{58}$$

$$r^{M-n} = b^{M-n} - A^{M-n}p_n^{M-n} \tag{59}$$

Equation 59 explains why using a linear combination of two different resolutions work in the V-cycle update.

$$\begin{aligned} A^{M-n}p_{n+1}^{M-n} &= b^{M-n} \\ A^{M-n}(p_n^{M-n} + prolong(p^{M-n-1})) &= b^{M-n} \\ A^{M-n}prolong(p^{M-n-1}) + \underbrace{A^{M-n}p_n^{M-n} - b^{M-n}}_{-r^{M-n}} &= 0 \\ A^{M-n}prolong(p^{M-n-1}) &= r^{M-n} \end{aligned} \tag{60}$$

If M is the initial resolution number for the grid, we can specify a lower minimum resolution number K. For example, we do not want to solve the pressure equations for grids with dimension sizes small as 1 and 2.

Algorithm 8 Multigrid method

```
for m = M − 1 down to K do
    φ^m = restrict(φ^{m+1})
    φ_s^m = restrict(φ_s^{m+1})
end for
for m = M down to K do
    Build sparse matrix A^m
end for
Build right hand side b
Initial guess p^M = 0
for i = 1 to N_Full cycle do
    Full cycle
end for
for i = 1 to N_V cycle do
    V cycle(M)
end for
```

In Algorithms 8, 9 and 10, we only use the restriction operators on the surface tracking level set and the residuals. The prolongation operator is only used on the pressure grids.

We know have all the components we need to solve the pressure equations and make sure the velocity field is divergence free. There are three variables we can tweak in the multigrid method for performance and convergence rate. The first one, N_{sweep}, determines the number of Red Black Gauss-Seidel iterations in each resolution. The other two, $N_{\text{Full cycle}}$ and $N_{\text{V cycle}}$, decide how many times to run each cycle method. The *Full cycle* converges faster

Algorithm 9 V cycle(m)

1: **for** i = 1 to N_{sweep} **do**
2: Gauss-Seidel to solve $A^m p^m = r^m$
3: **end for**
4: $r^m = r^m - A^m p^m$
5: $r^{m-1} = restrict(r^m)$
6: $p^{m-1} = 0$
7: V cycle(m-1)
8: $p^m = p^m + prolong(p^{m-1})$
9: **for** i = 1 to N_{sweep} **do**
10: Gauss-Seidel to solve $A^m p^m = r^m$
11: **end for**

Algorithm 10 Full cycle()

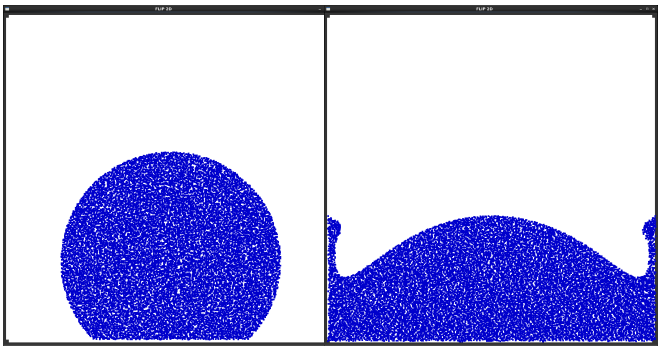
1: $p^{tmp} = p^M$
2: **for** m = K to M **do**
3: **if** $m \neq K$ **then**
4: $p^m = prolong(p^{m-1})$
5: **end if**
6: V cycle(m)
7: **end for**
8: $p^M = p^{tmp} + p^M$

in theory but in practice, all the restriction and prolongation operations can be expensive in an implementation. In general, the larger N_{sweep} is, the smaller can $N_{\text{Full cycle}}$ and $N_{\text{V cycle}}$ be and vice versa. The most optimal choice of parameters is situational and how to automatically detect the different scenarios is beyond this thesis.

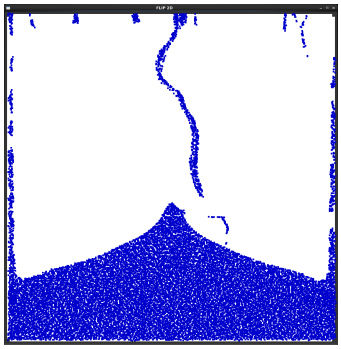
6 Results

A single threaded test application was programmed in C++ to evaluate the FLIP water simulation with multigrid as pressure solving method. No external libraries other than the C++ standard library was used in the simulation. The graphics library OpenGL was used to display the simulation where each particle in the FLIP simulation is represented as a blue quad. The simulation generated data for display 24 times per second.

The resolution of the grid in the test application was set to 128 for both N_x and N_y. The size of each side in the grid was set to $1m$ which gives each cells a width of $\Delta x = 0.0078$. As can be seen in the Figure 16a, the initial shape of the simulation is a half sphere. The solid wall boundaries are placed at the grid boundaries with one grid cell as the width. No extra solids to collide against were used. Parts of the simulation can be seen in Figure 16. More images of the multigrid simulation can be found in Appendix A.



(a) Frame 0. (b) Frame 15.

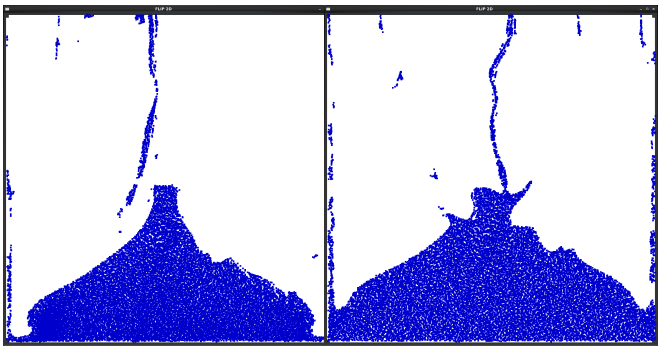


(c) Frame 60.

Figure 16: Results of the multigrid simulation.

The multigrid parameters used in Figure 16 are $N_{sweep} = 10$ and $N_{\text{Full cycle}} =$

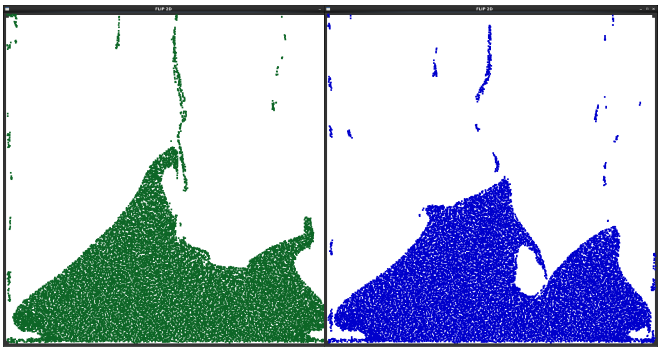
$N_{\text{V cycle}} = 4$. Figure 17 shows the difference between using a low number of cycles vs a high number of cycles. When $N_{\text{Full cycle}}$ and $N_{\text{V cycle}}$ are set to lower values, we see volume loss artifacts.



(a) $N_{\{\text{Full,V}\}\text{ cycle}} = 1$. (b) $N_{\{\text{Full,V}\}\text{ cycle}} = 8$.

Figure 17

A preconditioned conjugate gradient, PCG, pressure solver with incomplete cholesky factorization as the preconditioner was also implemented in the test application to test correctness. For information about the implementation, one can find a detailed explanation in [1]. A quick comparison between the multigrid method (with both cycles to be run 4 times) vs PCG can be seen in Figure 18. Although different pressure solving methods, they produce solutions that look similar. Images of the full PCG simulation can be found in Appendix B.



(a) PCG. (b) Multigrid.

Figure 18: A comparison at frame 70 between the PCG method (green) and the multigrid method (blue).

The results of the multigrid pressure solver shows that this method is sufficient for solving the pressure equations in a fluid simulation based on the

Euler fluid equations. The final two dimensional water simulation presented in this report looks convincing and was not implemented to satisfy any real-time restrictions. Traditional methods for solving the pressure equations produce similar results as the multigrid pressure solver.

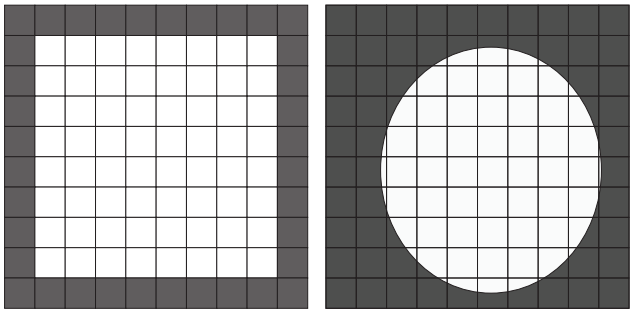
7 Discussion

7.1 Improving Boundary Conditions

In chapter 5 we assumed that the boundary conditions were always aligned perfectly with the grid cells. We also assumed that the velocity of all solids were zero. If we would allow solids to have a velocity $u^{\vec{solid}}$, Equation 48 has to be changed to

$$u^{solid}_{i+1/2,j} = u^*{}_{i+1/2,j} - \frac{\Delta t}{\rho}\frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{61}$$

It gets more complicated to set up the pressure equations when a cell is only partially covered with solid or air. Figure 19 shows two different cases, one that our simple approach covers very well and one that would lead to rectangular artifacts.



(a) regluar (b) irregular

Figure 19: Different solid boundaries.

Batty et al [7] presented a way to deal with irregular boundary geometry. It would be interesting to apply their ideas to the fluid multigrid solver implemented in this thesis. When setting up the pressure equations, we would need a way to find out how much the area next to an an edge is covered by fluid. The only change needed would be in Equation 45. If m is the previously mentioned fluid cover function ($0 \leq m \leq 1$), our linear system of equations need to satisfy the following:

$$
\begin{aligned}
(m_{i-1/2,j} + m_{i+1/2,j} + m_{i,j-1/2} + m_{i,j+1/2})p_{i,j} - \\
m_{i-1/2,j}p_{i-1,j} - m_{i+1/2,j}p_{i+1,j} - \\
m_{i,j-1/2}p_{i,j-1} - m_{i,j+1/2}p_{i,j+1} = \\
-\frac{\rho\Delta x}{\Delta t}(m_{i+1/2,j}{u^*}_{i+1/2,j} - m_{i-1/2,j}{u^*}_{i-1/2,j} + \\
m_{i,j+1/2}{v^*}_{i,j+1/2} - m_{i,j-1/2}{v^*}_{i,j-1/2})
\end{aligned}
\tag{62}
$$

7.2 Advecting the Level Set

We are constantly redefining the surface each step based on the position of the particles. This can lead to a noisy surface tracking. Another interesting option to try would be to create a smooth level set once and then advect the level set. The free surface, where $\phi = 0$ should be advected with the fluid velocity and we need to solve

$$
\frac{\partial \phi}{\partial t} + \nabla\phi \cdot \vec{u} = 0 \tag{63}
$$

Fedkiw and Osher [11] introduced high-order methods for solving Equation 62. If we advect both the level set and all particles every step, there will be cases where particles exists in regions where $\phi > 0$. What to do with those particles is not clear. One option could be to only advect the escaped particles in the extrapolated velocity field and not let them affect the pressure solve.

7.3 Parallel implementation

Solving for pressure is the largest and the most expensive computational part of the FLIP algorithm. Most fluid simulators have traditionally been using a preconditioned conjugate gradient pressure solver. A very popular preconditioner to use is the Incomplete Cholesky Factorization. Although making the conjugate gradient method to converge faster, this preconditioner cannot be run in parallel which is unfortunate since one has to apply the preconditioner before every conjugate gradient step. The multigrid approach has the big advantage that every step is easy to port from a serial implementation to a parallel one. Given a grid size of N_x and N_y, one can simply divide the grid into subregions proportional to the number of cores present in the architecture performing the computations. One of the benefits using the Red and Black Gauss-Seidel iteration scheme presented in Chapter 5 is that it does not matter in which order the subregions are updated in an iteration step

since we are guaranteed that all neighboring cells had their values updated in the previous step. The restriction and prolongation operators are also easy to perform in parallel since there are no read/write conflicts and once again the grid can be divided into subregions proportional to the number of cores available. Other trivial steps to run in parallel out of the items presented in the outline of the algorithm in Chapter 2 are:

1. Mark cells fluid
2. Apply external forces to grid
3. Create Level Set
4. Transfer grid velocities to particles
5. Advect particles

Transferring the particle velocities to the grid is harder to run in parallel because of the fact that two particles could potentially be writing to the same velocity in the grid at the same time and therefore introduce a write condition. This motivates us to use a special data structure for particles, storing them in different banks depending on their spatial position. Similar to the Red and Black Gauss-Seidel, we only update regions in parallel that are not neighbors and we can therefore assume that inside a thread there will never be a write condition.

Reinitializing the level set and extrapolating velocities outside of the fluid region are also a bit more complicated since these steps are both using the fast sweeping method. Jeong et al[16] explains an algorithm that solves the Eiokonal equation in parallel using a fast sweeping approach. Similar method can be used to extrapolate the velocities as well.

References

[1] R. Bridson, *Fluid Simulations for Computer Graphics.* 1st Edition, 2008.

[2] Y. Zhu and R. Bridson, *Animating Sand as a Fluid.* 2005.

[3] M. Mueller *A Multigrid Fluid Pressure Solver Handling Separating Solid Boundary Conditions.* 2011.

[4] H. Zhao, *A Fast Sweeping Method for Eikonal Equation.* 2005.

[5] J. Stam *Stable Fluids.* 1999.

[6] N. Foster and R. Fedkiw *Practical Animation of Liquids.* 2001.

[7] C. Batty, F.Bertails and R. Bridson *A Fast Variational Framework for Accurate Solid-Fluid Coupling* 2007.

[8] D. Enright, S. Marschner and R. Fedkiw *Animation and Rendering of Complex Water Surfaces.* 2002.

[9] F. H. Harlow *The Particle-in-Cell Method for Numerical Solution of Problems in Fluid Dynamics.* 1963.

[10] F. Harlow and J. Welch *Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface.* 1965.

[11] S. Osher and R. Fedkiw *Level Set Methods and Dynamic Implicit Surfaces.* 2002.

[12] A.J Chorin *Numerical Solution to the Navier Stokes equations* 1968.

[13] J.J Monaghan *Smoothed particle hydrodynamocs.* 1992.

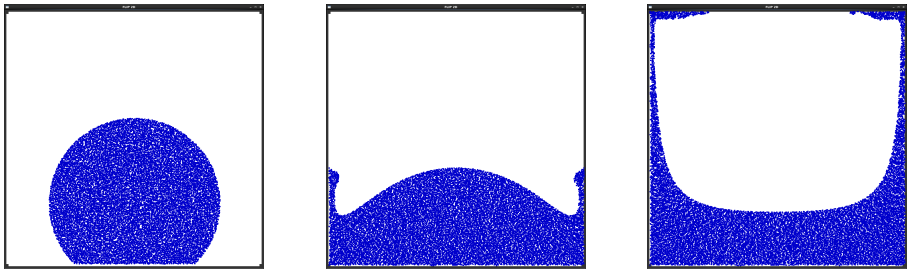
[14] M. Desbrun and M.P Cani *Smoothed particles: A new paradigm for animating highly deformable bodies.* 1996.

[15] N. Foster and D. Metaxas *Realistic animation of liquids.* 1996.

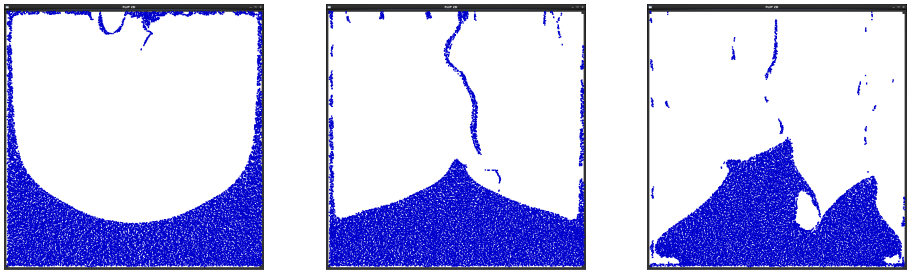
[16] W. K Jeong and T. Whitaker *A fast eikonal equation solver for parallel systems.* 2007.

[17] J. A. Baerenetzen and H. Aanaes *Generating Signed Distance Field From Triangle Meshes.* 2002.

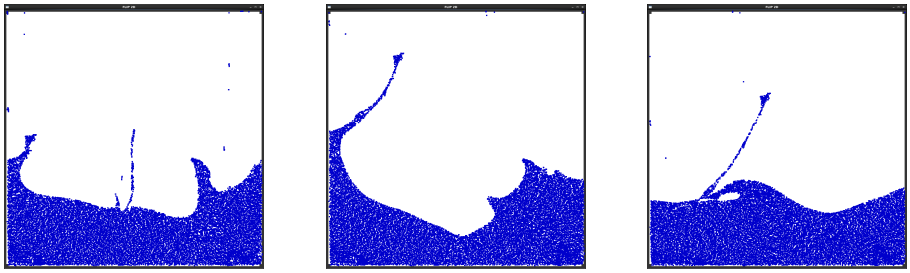
Appendix A: Multigrid Simulation



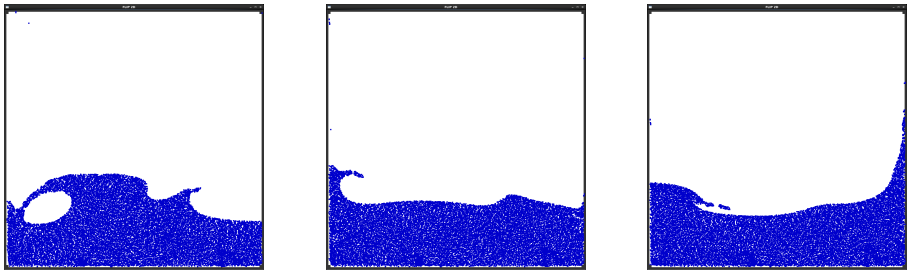
(a) Frame 0. (b) Frame 15. (c) Frame 30.



(d) Frame 45. (e) Frame 60. (f) Frame 75.



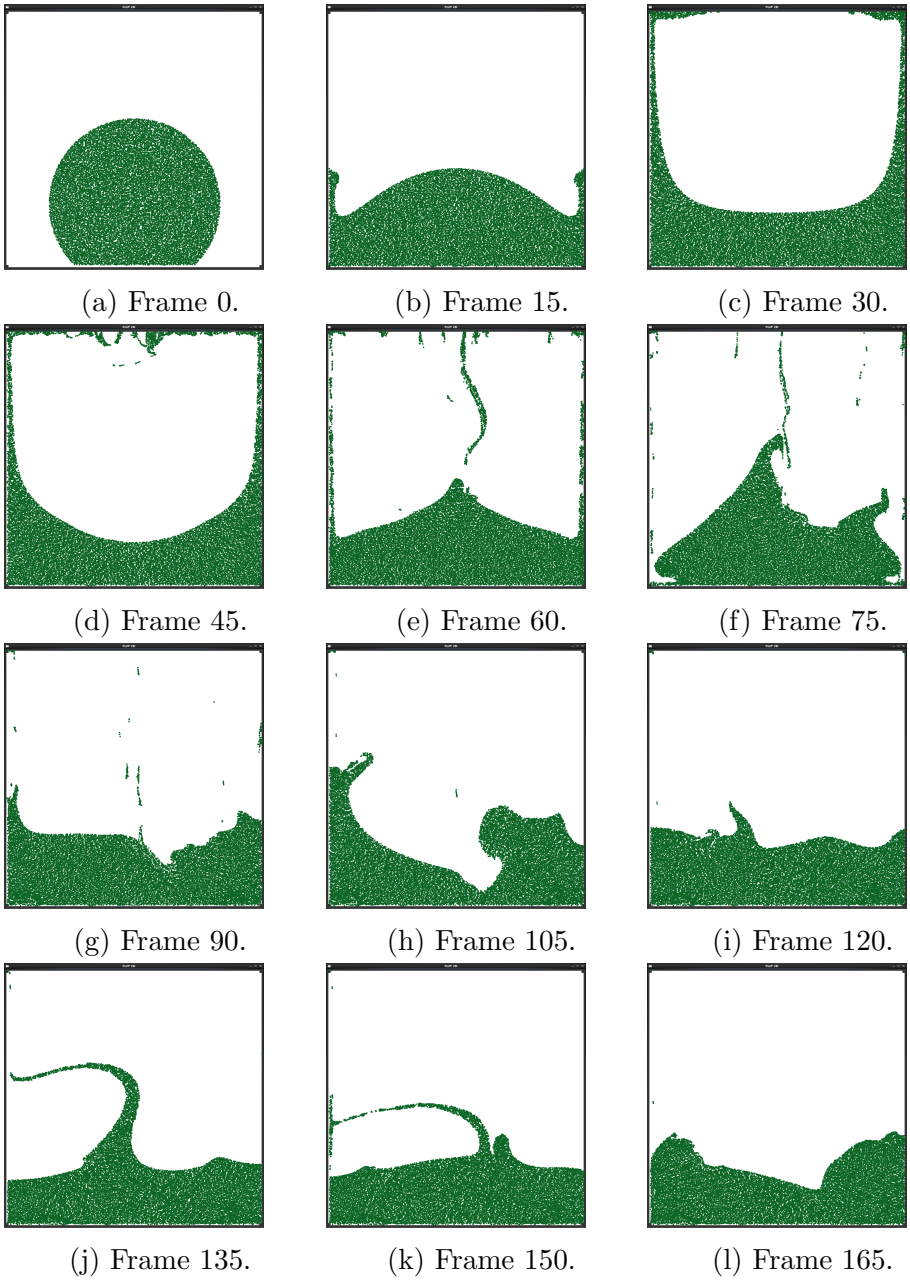
(g) Frame 90. (h) Frame 105. (i) Frame 120.



(j) Frame 135. (k) Frame 150. (l) Frame 165.

Figure 20

Appendix B: Preconditioned Conjugate Gradient Simulation



(a) Frame 0. (b) Frame 15. (c) Frame 30.

(d) Frame 45. (e) Frame 60. (f) Frame 75.

(g) Frame 90. (h) Frame 105. (i) Frame 120.

(j) Frame 135. (k) Frame 150. (l) Frame 165.

Figure 21