TU WIEN Informatics

Integrating GPU-based fluid simulation with Metaball rendering

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Konstantin Lackner
Registration Number 11716989

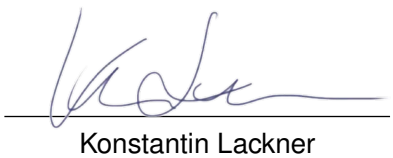
to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Bruckner

Second Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller

Vienna, 2nd February, 2022



Konstantin Lackner

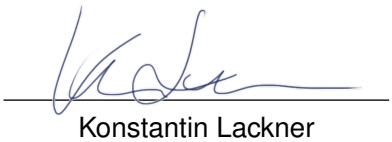
Eduard Gröller

Erklärung zur Verfassung der Arbeit

Konstantin Lackner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 02. Februar 2022



Konstantin Lackner

Acknowledgements

Many people helped me make this thesis possible. In this short section, I would like to thank all of them for their assistance with this project.

First and foremost, I want to thank my advisor, **Stefan Bruckner**, who did not only deepen my understanding of the matter considerably, but also helped me debug the code, as well as introduce me to scientific writing.

Secondly, I want to thank **Eduard Gröller**, the professor who introduced me to the idea of writing my bachelor's thesis during an Erasmus stay abroad and suggested me to my advisor and the relevant institute.

Furthermore, I want to express my deepest appreciation to my friend **George Tokmaji**, who helped me considerably in implementing the simulation described in this thesis, in particular, by providing his StdGLHelpers class.

Thanks also to GitHub user **bassicali**, who implemented his own version of the fluid simulation in inkbox and discussed his results with me at length.

I want to extend my gratitude to **Jordan Prior** for proofreading this text. As well as to **Theresa Bodner**, for proofreading my website that provides a subpage for this thesis. I also want to thank **Aaron Wedral**, **Jakob Lissy**, **Manuel Eiweck** and **Sabine Lackner-Feiler** for giving me feedback on the textual part of the thesis.

Lastly, I want to thank my family for enabling me to spend half a year abroad, working on this thesis.

Abstract

Fluid simulation is among the most researched topics in computer graphics. This is due to the fact that realistic fluid simulations are of first importance in many fields. The scope involves special effects in movies, animation in games, engineering and medical applications. Being one of the most researched topics of the field, there is a plethora of implementations and methods that have been discussed in extensive detail. Especially modern implementations focus on harnessing the GPU's advantage in calculating a multitude of operations at once, to create a realistic result while also maintaining solid performance.

This bachelor's thesis discusses fluid simulation on the GPU. More specifically, it implements a GPU-based simulation method and combines it with an implicit modelling procedure provided by my advisor, Stefan Bruckner. The goal of the thesis is thereby to provide a fast, sufficiently realistic fluid simulation and discuss its implementation, results and performance.

Kurzfassung

Fluidsimulationen gehören zu den meisterforschten Thematiken in der Computergrafik.Das liegt daran, dass Fluidsimulationen in verschiedensten Feldern von höchster Relevanz sind. Der Umfang der Anwendungsbereiche umfasst Special Effects in Filmen, Animation in Spielen, sowie mechanische und sogar medizinische Technik. Aufgrund dieser weiten Verbreitung gibt es bereits eine Vielzahl an Implementierungen verschiedenster Art. Besonders moderne Implementierungen sind darauf bedacht die auf einer GPU mögliche Parallelisierung möglichst gut auszunutzen, um realistische Simulationen zu erzeugen.

Diese Bakkalaureatsarbeit behandelt Fluidsimulation auf der GPU. Im Zuge dieser Arbeit wurde eine GPU-basierte Simulationsmethode implementiert und durch Implicit Modelling, welches mein Betreuer, Stefan Bruckner, in seinem Programm Dynamol zu Verfügung stellt, visualisiert. Das Ziel der Arbeit war es, eine schnelle, ausreichend realistische Simulation zu erzeugen und deren Implementierung, Resultate und Performance zu untersuchen.

Contents

CHAPTER 1

Introduction

As an avid enjoyer of video games and movies, I had been looking to write my bachelor thesis in the field of Computer Graphics. When Stefan Bruckner presented this topic to me and admittedly after slight hesitations concerning the difficulty of the task ahead, I jumped at the opportunity to create this project under his supervision.

1.1 Problem Statement

Realistic fluids are of great importance in a plethora of Computer Graphics applications. Examples of this can be found all throughout cinema history as well as in the domain of computer games. Of course, fluid simulation is not only relevant for fully animated movies. The realistic simulation of fluids plays a role in many a visual effect too. Obvious examples of this are movies with big natural disasters such as "The Day After Tomorrow" (cf. Figure 1.1). Fluids, including lava, smoke, gas and of course bodies of water that play a key role in the movies, are simulated using fluid simulation programs. But even smaller effects such as cigarette smoke, muzzle flashes and the like behave in a fluid-like manner and hence can be modelled accordingly.

Another domain that is highly dependent on realistic fluid simulation is engineering. Commonly referred to as Computational Fluid Dynamics (CFD), fluid simulations are used to model different kinds of flows such as water, heat or air either in or around components. One such an example is CFD in turbomachinery, i.e. internal and external fluid flows in turbines (cf. Figure 1.2) [PAD+17].

Yet another example of application fields are reacting flows and combustion. This part of the field is specialised on combustion engines and chemical reactions in fluids. Predicting the behaviour of fluids on a chemical level is important in the construction as well as testing of motors to prevent e.g. material failure due to overworking of key parts. One such example would be the calculation of thermal stress on said parts.

Figure 1.1: Flood wave in "The Day After Tomorrow" (Image taken from [The])

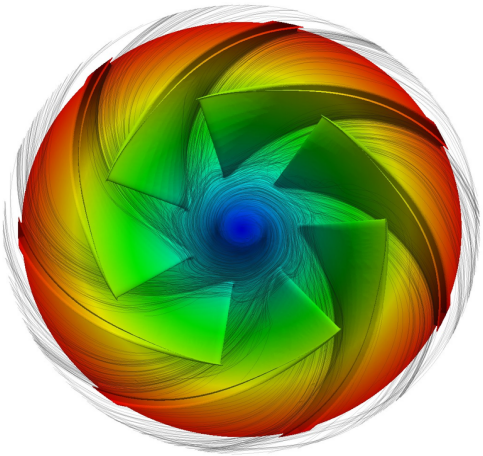


Figure 1.2: Flow in a CFD turbomachinery simulation (Image taken from [cfd])

With all the applications mentioned above, one of the biggest challenges in fluid simulation is the combination of achieving visually pleasing, i.e. realistic, results while keeping the computational power needed to achieve them to a minimum. From an implementation standpoint, this, of course, quite evidently asks for GPU implementation, as I will elaborate on in later chapters. Many prior implementations of fluid simulations suffer from the limitations of their respective time. Nowadays, however, the possibility to implement established methods on the GPU in e.g. OpenGL is not only a conceptual deliberation but has been implemented in various forms.

Goals for this work

Building on Stefan Bruckner's Dynamol [Bru19], an implicit modelling approach for molecular surfaces, the goal of this thesis was to implement a fluid simulation for the

given Metaball-based rendering . The challenge in developing such a simulation was that it had to be fully GPU-based. Dynamol's implicit modelling approach is executed in parallel on multiple threads, hence, including a fluid simulation, that simulation also had to be executable in parallel.

The thesis' implementation is largely built on the foundation laid by Jos Stam's and in further consequence Mark J. Harris' work in "Stable Fluids" [Sta99] and "GPU Gems" [Fer04] respectively. Coming back to the relevance of fluid simulation in media: Jos Stam was awarded three technical achievement awards, respectively in 2006, 2008 and 2019 [Uof]. The awards are issued by the Academy of Motion Picture Arts and Sciences, and are more commonly referred to as "Oscars".

The mathematical model used to simulate fluids is the same as the one presented by Stam and Harris [Sta99, Fer04], the implementation, however, is considerably modernised regarding both but primarily Stam's work.

CHAPTER 2

Related Work

In this chapter I present a broad overview of existing literature and knowledge on the topic of fluid simulations and more specifically GPU-based simulation techniques to draw a conclusion bridging the gap to the implementation at hand in this thesis.

Realistic fluid simulations and animation are of utmost importance in various fields of application. From computer games to medical applications or engineering software, the need for realistic looking and/or behaving computer generated fluids has brought forth a plethora of implementations of such simulators. At the very core of all of them sit two mathematical equations initially developed in 1820; the Navier-Stokes equations. I will go into the details of the equations before long, essentially, however, they model how a fluid moves. Inputting an initial state and forces the equation will solve for the state to be expected in the next time-step.

2.1 Fluid Simulation

I will cover three popular approaches in the simulation of single fluids, i.e. models of homogenous fluid flows as opposed to heterogenous simulations with different miscible or immiscible fluids, all of which incorporate the Navier-Stokes equations in their calculations. There are, of course, also hybrids combining several of the three approaches covered in this thesis:

1. **Mathematical model-based approaches:** Simulations that rely on mathematical models to visualise fluid flows.
2. **Particle-based approaches:** Simulations that save forces into particles and calculate their interactions.
3. **Grid-based approaches:** Simulations that divide flows into grids assigning values to grid cells instead of the fluids themselves.

2.1.1 Mathematical model-based approaches

State of the art implementations of mathematical model-based approaches include the Lattice-Boltzmann model (LBM) [TR04]. The approach is a strictly mathematical model-based fluid simulation but can be thought of as a combination of grid-based and particle-based approaches. The fluid is modelled as a set of particles on a discretised lattice, propagating and colliding in the algorithm's steps.

Spectral methods are another such example [Can88]. They may be viewed as an extreme development of the class of discretization schemes for differential equations known generally as the method of weighted residuals (MWR) [FS66]. The MWR stems from methods used for boundary-value problems. It separates an approximation for a differential equation into basis functions with arbitrary parameters which can then in turn be adjusted to converge to a solution [FS66]. The differential equations solved are of course the Navier-Stokes equations governing fluid motion.

This thesis' implementation, however, is a hybrid of grid-based and particle-based models and hence the thesis will not go into further detail about mathematical model-based approaches. For a deeper analysis of such approaches, refer to spectral methods by Canuto [Can88].

2.1.2 Particle-based approaches

Particle-based models, instead of relying on grids, simulate a Lagrange viewpoint through a cloud of particles, i.e. a cloud dependent on time, its elements and the velocities affecting them, rather than time and points in space (cf. Eulerian viewpoint). This approach intrinsically conveys a straightforward solution to the first Navier-Stokes equation. Assigning a mass and velocity to every particle, the conservation of mass is simplified to conserve the number of particles in the system and aids in simulating interactions between them. The interactions between particles can be modelled in various forms. One such approach is to model particle behaviour according to the Lennard-Jones potential, as proposed in Miller and Pearce's Globular Dynamics [MP89]. Fundamentally, the Lennard-Jones potential describes the attractive and repulsive forces acting between two particles. Particles further apart will experience a weak force of attraction, the closer they get to each other, the stronger the repulsive force acting between them becomes. Lucy [Luc77] first introduces the Smoothed Particle Hydrodynamics (SPH) method, i.e. moving particles according to hydrodynamic and gravitational forces, achieving thereby a simulation model for astrophysical phenomena. In the most basic form of the SPH and WCSPH (Weakly Compressible Smoothed Particle Hydrodynamics) algorithms, the forces acting on a particle are summed up as weighted contribution from N neighbouring particles. This method was later improved upon by Desbrun and Cani-Gascuel [DCG98].

The implementation of strictly particle-based approaches like the ones mentioned in this chapter are well beyond the scope of this thesis. For a more thorough understanding of the methods, please refer to the aforementioned works of Miller and Pearce as well as Lucy, linked in the bibliography [MP89, Luc77].

Compressibility of fluids

The SPH approach models compressible fluids, however, the incompressibility of fluids is a major factor in modelling fluid-like simulations. Strictly speaking, of course, in reality, no fluid is incompressible. However, the forces needed to compress fluids strongly vary depending on the fluid in question. Air for example is easily compressed with the application of a minimum amount of force. Water on the other hand requires great amounts of force to be compressed. In everyday life, most liquids exhibit somewhat similar behaviour, hence, incompressible flows model fluids more realistically than compressible flows. To account for this, several methods to implement incompressible SPH approaches have been proposed. Raveendran et al. [RWT11] present a way of enforcing incompressibility by combining the SPH approach with a Poisson Solver on a coarse grid. This method could hence be defined as a hybrid approach. Combining particle-based with grid-based methods and thereby achieving incompressibility is a popular approach in fluid simulation. The simulation method chosen for this thesis is, in its core, a grid-based approach but also makes use of particle-flows for simulations; bringing with it a compressibility issue that will be gone into in detail in Chapter 4 and 5 of this thesis.

2.1.3 Grid-based approaches

Grid-based fluid simulations make use of the Eulerian viewpoint rather than the Lagrangian one, as discussed in the introduction to particle-based approaches. Instead of tracking masses and velocities of particles, the system relies on definite points in space at which values such as velocity or density are stored and read.

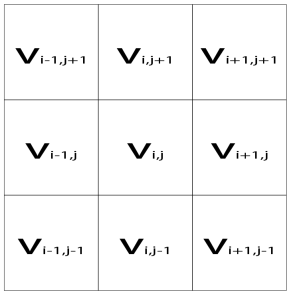


Figure 2.1: 2D grid with each cell having a 2D vector as velocity

Velocities and densities in the fluid are sampled in a grid as illustrated in Figure 2.1. Smooth movements are achieved by interpolation between grid cells.

This results in a susceptibility to mass loss but also improves the numerical accuracy as working on a spatial derivative is much more precise than tracking particles in unstructured Lagrangian clouds [BMF, p.7]. Furthermore, grid-based simulations gain quite considerably from parallelisation on the GPU. Grid-based fluid solvers have to calculate a considerable number of cells in every iteration and hence benefit greatly

from accessing and updating cells asynchronously [AG21]. This holds especially true for simulations with uniform Cartesian grids, like the one used in this thesis' implementation.

2.2 Groundwork for this thesis

The simulation method chosen for this project is based on Stam's stable fluids method [Sta99] introduced in 1999 and his successive work [Sta03]. Stam describes a two dimensional version of the simulation, mentioning that a three dimensional implementation is possible. Stam's simulations were, however, as was usual at the time, CPU-based and hence not suitable for parallelism. A two dimensional GPU-based approach was introduced by Mark J. Harris [Fer04] in 2004. Building on this, Crane et al. [Ngu07] presented a three dimensional GPU-based version in 2007. Both versions were based on the now deprecated Cg framework. Inkbox by the Github user bassicalli [Ink] provided an OpenGL and GLSL implementation that this thesis is directly based on. Many of the shaders are directly taken from there. The novelty in this thesis lies in its method of visualisation. While the aforementioned implementations visualise their simulations by directly rendering 2D or 3D forces as RGB colours, this thesis uses Stefan Bruckner's Dynamol [Bru19] to render Metaballs. Hence, one of the main challenges in the implementation part of this thesis was to implement a Transform Feedback as a linking layer between simulation and visualisation.

The intention of Stam's stable fluids was to address instability of simulations in earlier fluid solvers [AG21, Sta99]. For large time-steps, earlier solvers tended to "blow up", meaning the numerical scheme produced, when sufficiently stationary, overflows, where the simulated quantities exponentially grew to finally overflow. This happens when solving for the advection step, the step that computes the force transmission in and between grid cells. *"The evolution of these quantities over time is given by the Navier-Stokes equations."* [Sta99, p.2].

2.2.1 Navier-Stokes equations

The Navier-Stokes equations describe fundamental physical laws, modelling fluids. They are at the core of almost every fluid simulation [Sta99, Sta03, Fer04, Ngu07].

u ... fluid velocity (field)

p ... density

v ... kinematic viscosity of the fluid

f ... external force

$$\nabla \cdot u = 0 \tag{2.1}$$

$$\frac{\delta u}{\delta t} = -(u \cdot \nabla)u - \frac{1}{p}\nabla p + v\nabla^2 u + f \tag{2.2}$$

Equation 2.1 describes mass conservation. Moving a fluid, not subtracting or adding to it, the fluid's mass is constant. Equation 2.2 is essentially Newton's second law, saying that mass * acceleration is force. They are hence usually labelled Conservation of Mass equation and Conservation of Momentum equation respectively. For the second equation, the left-hand side, $\frac{\delta u}{\delta t}$, is Newton's second law, while the right-hand side describes, for the first two parts, $-(u \cdot \nabla)u - \frac{1}{p}\nabla p + v\nabla^2 u$, the internal forces in the fluid and for the third part, f, the external forces such as gravity. Fundamentally, the Navier-Stokes equations are well understood, however, it is yet to be proven whether either always smooth, globally defined solutions exist, or the equations potentially break, i.e. there are cases in which no solution exists [Fef00]. The Navier-Stokes equations are used universally in modelling all kinds of fluids. Applications encompass flow-visualisations as well as medical visualisations, graphics for computer games, or industrial calculations involving fluids. This makes the Navier-Stokes equations and the question whether a smooth solution always exists one of the seven Millennium Price Problems. The exact statement of the problem was formulated by Charles Fefferman [Fef00].

For a time-step in a fluid simulation, a convergent solution for the Navier-Stokes equations has to be found. Stam [Sta99, p.2] describes this as follows: "The pressure and the velocity fields which appear in the Navier-Stokes equations are in fact related. A single equation for the velocity can be obtained by combining Eq.1 and Eq.2.". The solving of this single equation is split into four steps, Force Addition (Impulse), Advection, Diffusion and Projection, which will be gone into in detail in Chapter 3 and 4 of this thesis. The second step, Advection, which describes a propagation of forces in the fluid, is modelled by the term $-(u \cdot \nabla)u$, the non-linear term in the Navier-Stokes equations. In previous solvers such as Foster and Metaxas [FM97], this was achieved by using finite differencing, resulting in these kind of solutions only being stable for small time steps and/or large velocities.

2.2.2 Stability

As mentioned before, stability issues are a big concern for fluid simulations. The novelty in Stam's work, which is then in turn also used in almost all grid-based solutions leading up to this implementation, is the use of the method of characteristics, a technique to solve partial differential equations. Stam [Sta99, p.3] explains the method: "At each time step all the fluid particles are moved by the velocity of the fluid itself. Therefore, to obtain the velocity at a point x at the new time $t + \Delta t$, we backtrace the point x through the velocity field w_1 over a time Δt. This defines a path $p(x, s)$ corresponding to a partial streamline of the velocity field. The new velocity at the point x is then set to the velocity that the particle, now at x, had at its previous location a time Δt ago".

This results in a maximum value for the resulting field as described in Equation 2.3. The value cannot exceed its predecessors', thereby making it inherently stable.

$$w_2(x) = w_1(p(x, -\Delta t)) \tag{2.3}$$

Inherent stability as proposed by Stam [Sta99, p.3]

Another „blowing up“-related, i.e. concerning the tendency of the scheme to produce overflows for certain values, problem Stam solves in his implementation has to do with the third step in the simulation, the diffusion step. The diffusion step solves for the effect of viscosity [Sta99, p.3] and can be solved using an explicit time step. This, however, again, proves to be prone to becoming unstable for large viscosities. Therefore, Stam implements an implicit method instead, resulting in a sparse linear system. The fourth step, projection, also produces a sparse linear system. The former of the two, the diffusion step's sparse linear system, is solved using Gauss-Seidl relaxation, whereas the latter, the system stemming from the projection step, is solved by backtracking through the density grid and interpolating densities from neighbouring fields, or how Stam [Sta03, p.7] puts it: *"Similarly to the diffusion step we could set up a linear system and solve it using Gauss-Seidel relaxation. However, the resulting linear equations would now depend on the velocity, making it trickier to solve."* Stam overcomes these issues by modelling densities as particles. Particles can be traced through space and time, tracing through space, however, proves difficult in the grid system of the simulation, as for a final result, particles would have to be converted back into grid values. Instead of trying to accumulate particles in cells, Stam chooses to trace the particles back in time one time step and observe only the particles that end up exactly in cell centres as illustrated in figure 2.2. The density in such particles can be interpolated from their neighbours at their starting position.

Harris' GPUGems2004 implementation and by extension the one in this thesis, solve for diffusion using Jacobi relaxation instead of Gauss-Seidl relaxation. In most other regards, the simulation in this thesis is similar to the one presented by Harris. The visualisation approach uses implicit modelling and is hence substantially different from the works of Stam [Sta99] and Harris.

2.3 Implicit modelling

Implicit modelling relies on mathematical tools to automatically derive models from data. As opposed to explicit methods, implicit methods spare the laborious work of defining models by hand.

The visualisation in the implementation of this thesis is provided by Dynamol which operates on Metaball Rendering [Bru19]. Metaball Rendering, originally introduced by Blinn in the early 1980s and referred to as blobs or blobby objects back then, is an implicit modelling method initially used to model atom interactions [Bli82]. In Dynamol, Metaball Rendering is used to represent molecular surfaces. Metaballs can essentially be considered as particles surrounded by density fields. Figure 2.3 shows 2D Metaballs visualising density with decreasing opaqueness. The density decreases with distance from the centre as can be observed in Figure 2.3. The purple centres are surrounded by density

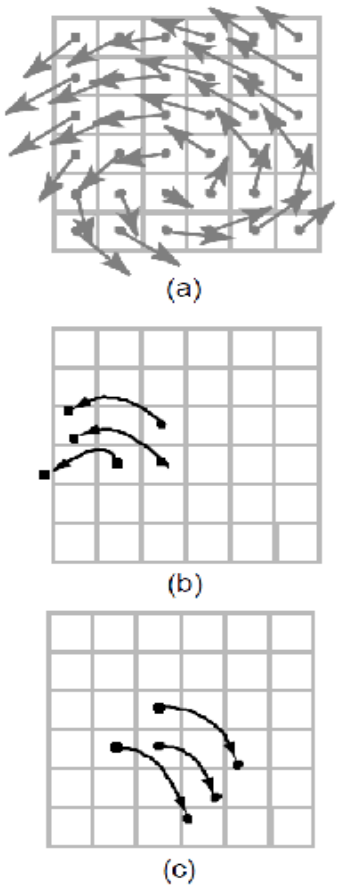


Figure 2.2: Tracing particles backwards through time to work with the particles that end up exactly in cell centres (Image taken from [Sta99] (modified, rearranged))

fields, their respective densities are illustrated with differing opaqueness, decreasing outwards from the centre.

In the rendering process, a ray is cast through the volume, checking densities at all points. Together with a density threshold, different isolines can be created cf. Figure 2.4. Depending on those edges, different shapes can be established.

Based on the threshold, a surface coating the particle can be established as seen in Figure 2.5. This creates the typical "blobby" shape of connected spheres, i.e. connected circular surfaces. These can be used to model fluids due to their resemblance of the effects observed in liquids due to cohesion.

The Metaballs melt into each other, creating a contiguous object. An arbitrary point's density can be determined by simply summing up both Metaballs' influence, i.e. summing up their densities, matching the density sums against the thresholds. This assures a smooth blending on edges between two Metaballs.

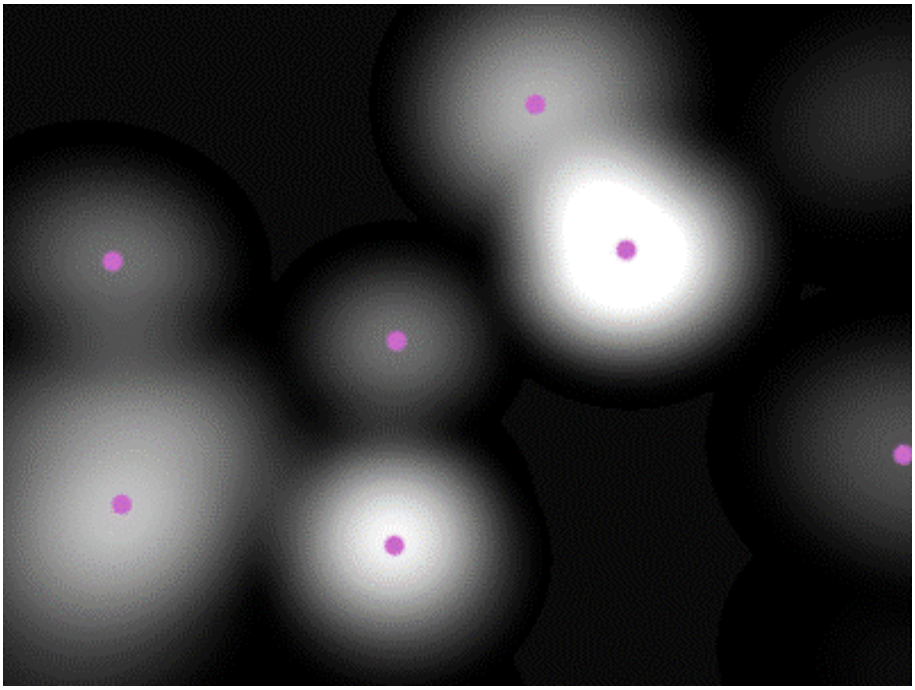


Figure 2.3: Decreasing densities around centre points in 2D (Image taken from [Met])

Choice of decaying function

An essential consideration with Metaballs is the choice of a decaying function used to calculate the influence of a particle on an arbitrary point. While Bling uses exponentially decaying functions and a Gaussian bump as the density function, Wyvill et. al. later improved upon this, using a cubic polynomial based on the radius of influence [WMW86]. This accelerates the process quite considerably. A function that falls after a certain point inherently makes the calculations for each raycasting step easier. In a scenario with 1000 particles, at each step along the ray, the sum of 1000 functions would have to be calculated. Having an influence of nought after a certain point, in most cases, will simplify the calculations tremendously.

Further simplifications

Another simplification which is also implemented in Dynamol, is interleaving surface intersections as well as sorting, i.e. the algorithm simply terminates at the first surface hit instead of going through all surfaces. Thereby it also avoids the need to sort a list of intersections [Bru19]. It has been suggested in prior work, that polynomial approximation of Gauss surfaces (such as the ones used in Dynamol) could simplify the process of finding surface bounds, this, however, depends on the use case. Dynamol for example, values the appearance and properties of real Gaussian surfaces above the simplicity of just approximating them [Bru19, p.5].

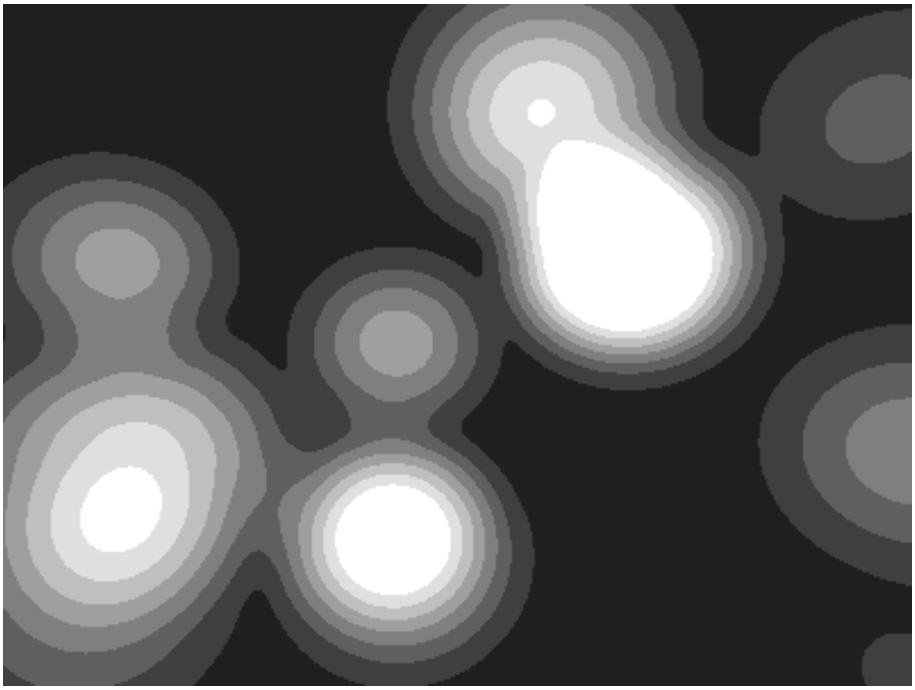


Figure 2.4: Possible borders for different thresholds in 2D (Image taken from [Met])

2.4 GPU Implementation

The method described earlier, i.e. a combination of the works of Stam [Sta99], Harris [Fer04] and Bruckner [Bru19], is naturally, like its respective parts, perfectly suitable for GPU implementation. The Raycasting involved in finding surfaces for Metaballs is a typical Shader use case. Dynamol implements its Metaballs, as is common practice, on the GPU [Bru19]. Therefore, to achieve maximum efficiency, the fluid simulation described earlier in this chapter should also be implemented on the GPU. This serves the purpose of not having to read from the main memory in between simulation steps. The whole simulation runs on the GPU and can hence be run without any interruptions e.g. communication between CPU and GPU.

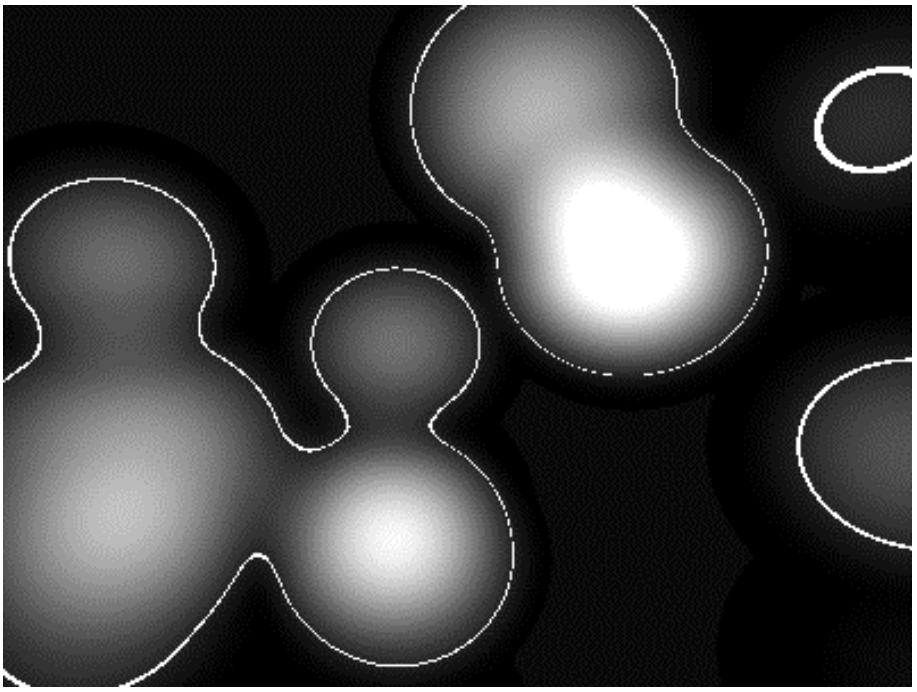


Figure 2.5: Formed isolines in 2D (Image taken from [Met])

CHAPTER 3

Method

In this chapter I give an outline of the general methods used to build the simulation approach proposed in this thesis. The technical details are elaborated on in the Implementation part.

The simulation presented in this thesis is based on the stable fluids approach first described by Jos Stam in 1999 [Sta99]. Like in Stam's method, a discretised grid of vectors representing the forces that describe the fluid flow is calculated. The simulation described is not a physically accurate simulation of fluids but rather a visually sufficiently convincing one. As mentioned in Chapter 2, physical accurateness, in this scope, is less important than visually pleasing results. For applications such as video games, movies and in some cases even scientific applications, fluids have to look and behave like real-world fluids but do not necessarily have to satisfy all their physical behaviours.

Many of the figures presented in this chapter are sets of frames from a video capture of various simulations. For a better understanding, see the video version of them on my website's subpage dedicated to this thesis [Kon].

3.1 Method outline

The simulation approach presented in this thesis uses a 3D texture, the velocity texture, to store the fluids velocities, i.e. forces that act in the fluid, as 3D vectors. The texture can be regarded as a grid. Its dimensions define the grid's resolution. The bigger the texture, the finer the resolution of the grid. The velocities in neighbouring grid cells affect each other. Flows move through the grid, each cell's velocity directly affects the velocities in the neighbouring cells, as well as a second texture, the pressure texture. Forces that propagate through a fluid build up pressure in the fluid. In real fluids, molecules would be pressed against each other, accelerating each other in the process.

This is described in Newton's second law, and is, as mentioned before, the statement of the second Navier-Stokes equation cf. Equation 2.2.

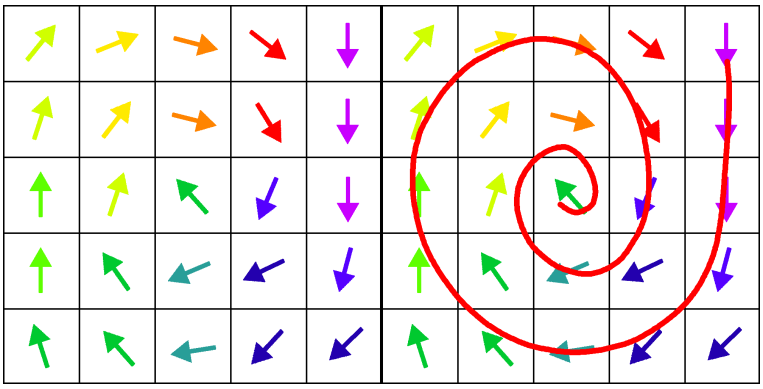


Figure 3.1: 2D visualisation of the grid with force vectors, colour-coded to the corners of the field: top left = green, top right = yellow, bottom right = red, bottom left = green

The simulation itself does not use particles but rather only works on the grid values, simulating the flow in terms of velocities in the cells as seen in Figure 3.1. This can be directly visualised by attributing colours to the cells' vectors. An ex-ante prototype for this simulation illustrates the velocity vector field of the simulated fluid in a 800x800 grid (see Figure 3.2). An impulse is added by dragging the mouse cursor through the window. The impulse force is then advected and diffused through the fluid. The colours in this figure represent the forces' properties, the more opaque a colour, the stronger the force – hence, a mitigating effect can be observed in the waves spreading from the input. The forces are also colour-coded. A force towards the upper right corner is yellow, likewise, clockwise, the other corners are red, blue and green.

Dynamol provides the particles for the particle-based part of the simulation [Bru19]. The molecule particles, which are visualised using Metaballs, are moved through the grid and accelerated along the vector in every grid cell they pass through. As opposed to classic particle-based approaches, the particles themselves store no information on velocitiy or other forces in this simulation. The moving of the particles is achieved solely through changing the Metaballs' centre positions.

3.2 Simulation steps

The simulation created in the course of this thesis is divided into five major steps that are performed for each time step of the simulation to achieve a realistic looking fluid flow. All of the steps have a different purpose and have to be run through in a specific consecutive order

The steps are:

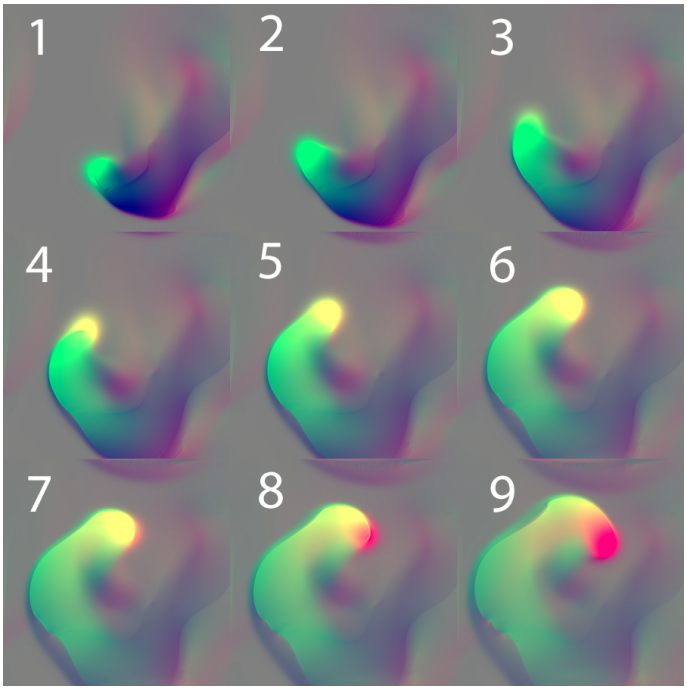


Figure 3.2: Visualising the flow by attributing colours to the forces as before in Figure 3.1, this figure shows nine captures of a flow occurring in the order of their respective numbers (Video)

1. **Force Addition (Impulse):** This step adds an impulse either as random droplet being dropped into the fluid system, i.e. a circular spread of forces, or via user input.
2. **Advection:** This step moves the velocities through the fluid.
3. **Diffusion:** This step attributes for the resistance of the fluid to flowing. The higher the viscosity of the fluid, the stronger the diffusion.
4. **Boundary Calculations:** This step models the border of the volume the fluid is in.
5. **Projection:** This step forces the fluid to be mass conserving as described in the first Navier-Stokes Equation 2.1.

3.2.1 Force Addition (Impulse)

This is the input step in which forces are added to the simulation. Through this step, the user can interact with the fluid simulation, creating force impulses in the system. In the early 2D version, this is either achieved manually via user input, i.e. by dragging the mouse through the visualised liquid, or by activating what Github user bassicalli names

"droplets mode" in his work Inkbox [Ink]. When manually adding a force to the system, the force added is a constant velocity added in the direction of the drag. The droplets mode generates random positions and applies uniform, radial forces at them. This results in a series of droplets in the force field, as illustrated in the 2D ex-ante version in Figure 3.3. In the final 3D version, force can be added into the system by defining a vector direction over two points and setting a magnitude in a sub menu as seen in Figure 3.4. The 3D version also includes a droplets mode.

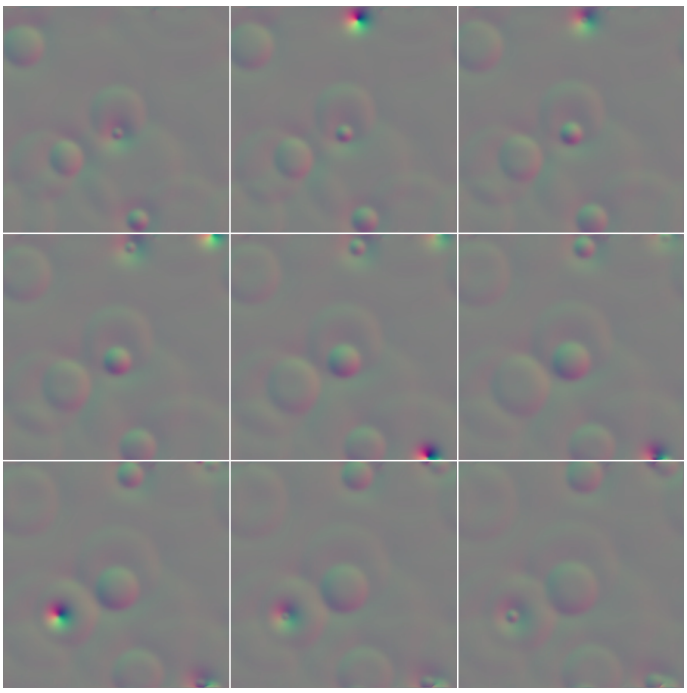


Figure 3.3: Droplets mode in an early 2D version of this thesis' simulation adapted from [Ink] - Uniform, radial forces are applied at randomised points in the system, the colour coding corresponds to the one explained in Figure 3.1

3.2.2 Advection

This step applies the flow forces to the fluid and thereby to the forces themselves. In this step the forces propagate through the medium. This is where the backtracking of particles in the system discussed in Chapter 2 comes into play. The value for a grid cell is determined by back tracing a fictive particle one time step and interpolating the values closest to the back traced point cf. Figure 3.5 and 2.2. This implicit method replaces explicit approaches such as Runge-Kutta or the midpoint method for calculating the influence of the forces next to the current grid cell, and thereby prevents the simulation from becoming unstable for large time steps as explained by Stam [Sta99, p.3].

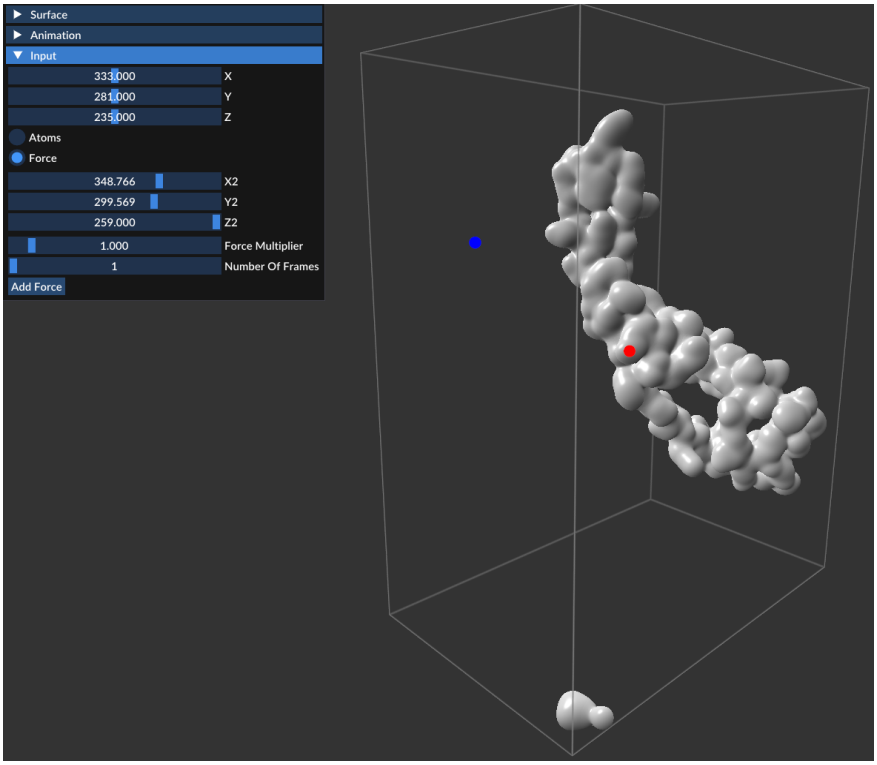


Figure 3.4: Defining a vector in the final 3D version

3.2.3 Diffusion

This step accounts for the loss of momentum in the fluid caused by its viscosity. After this step, the fluid's forces are mitigated according to its viscosity, i.e. the viscosity of the fluid thwarts its movement, until, after a certain number of diffusion steps, the movement force is extinguished (or hits a volume border). This of course makes sure the fluid behaves according to the left side of the second Navier-Stokes equation, Newton's second law cf. Equation 2.2 as well as the term $v\nabla^2 u$ on the right-hand side.

3.2.4 Boundaries

The boundary step encloses the fluid in its volume and simulates the border conditions. Grid cells at the very edge of the volume, e.g. a side of the box the fluid is contained in, are treated differently. The boundary shader goes through the texture and checks if a cell is on the edge of the volume in any direction. If a cell is on an edge, the shader takes the cell's velocity vector, changes the respective component (e.g. the y component for the boundary on the bottom of the volume) to nought and writes it back to the cell.

This creates an effect, where velocities, unlike in real systems, are expunged without any form of decline or counter force.

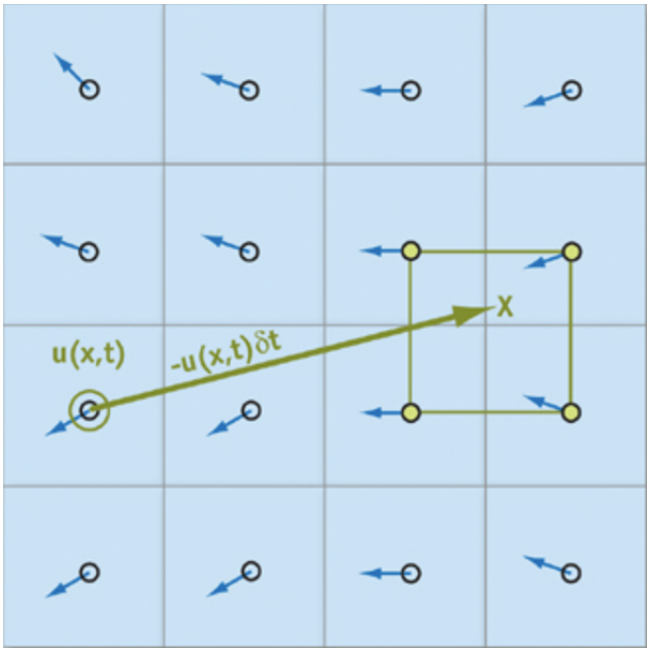


Figure 3.5: Back tracing a grid cell's centre one time step (Image taken from [Fer04])

Eulerian and Lagrangian approaches

The border condition discussed in the last subsection introduces an error into the system. Where in nature, fluids would bounce off of walls, exhibiting a ricocheting behaviour, this method only stops the simulated fluid from escaping its bounds. This border condition is a necessary compromise due to the Eulerian aspect of the simulation. Where in a Lagrangian system, particles carry a force and mass, the Eulerian grid approach does not allow for separately handling single particles. Velocities are only saved in the grid and not related to specific particles. In a Lagrangian simulation, a particle, when hitting a boundary, could receive a counter force, acting against its prior velocity. This is illustrated in Figure 3.6 with the example of a non-elastic particle hitting the floor boundary. The particle, having a mass, can bounce off the boundary. After the bounce, the force pointing upwards is reduced due to inertia, while the gravitational force pulling the particle down stays the same. After a few bounces, the force is reduced to a point where the particle rests on the boundary.

Introducing a boundary counter force in an Eulerian system, however, results in unwanted behaviour. A counter force pushing inside the volume rather than outside of it has to be placed in the last grid rows/columns resulting in a grid like illustrated in Figure 3.7. For cells that function as edges for multiple sides, the force is flipped in all respective directions.

This, however, produces the behaviour illustrated in the first row of Figure 3.8. Having a counter force in an Eulerian grid, results in the particles jumping around between the

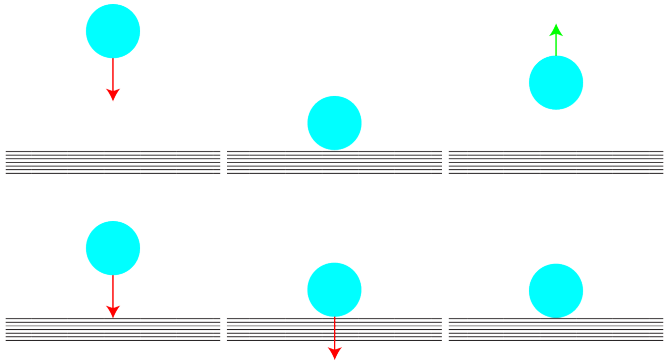


Figure 3.6: Lagrangian border conditions - The upper three images illustrate a counter force after hitting a wall, whereas the three images on the bottom represent the actual border condition, where hittin a wall simply reduces the particles' velocity to nought

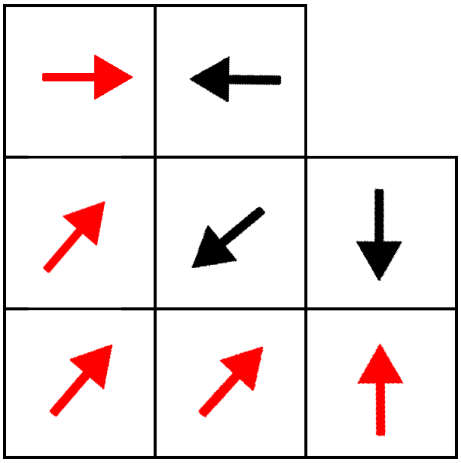


Figure 3.7: Cells on the edge of the volume (colour coded with red forces) flip velocities and thereby act as counter forces

last and second to last grid row next to a border. As there is no mass in the system, inertia cannot contribute to reducing the forces that act on the particles. Hence, the gravitational pull as well as the counter force are never reduced and keep acting against each other. The second row in Figure 3.8 illustrates the solution for this issue. Instead of having a counter force in the last row before the boundary, the boundary condition simply changes the velocity in that position, in its relative component (e.g. y for a velocity next to the ground boundary cf. Figure 3.8) to nought. This prevents bouncing,

as would be expected in a real system but also prevents the unwanted jumping around between the respective grid rows. A possible solution to make a realistic boundary that is still in accordance with the Eulerian grid system, would be to introduce a broader boarder zone. In this border zone, a new shader would handle particles, according to their entry velocity, i.e. the velocity they enter the boarder zone with. This solution, while theoretically possible, goes well beyond the scope of this thesis but provides an example of possible future improvement.

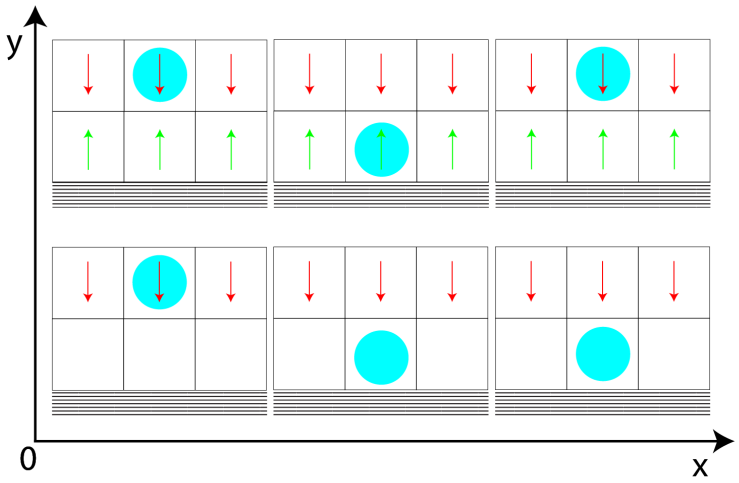


Figure 3.8: Comparison of solutions for an Eulerian grid boundary condition, the top three images illustrate the problem of flipping forces on the border of the volume, the second row illustrates the solution chosen for the final version of the simulation

3.2.5 Projection

The projection step is separated into two parts. Its goal is to subtract a gradient from the velocity field to make the velocity field incompressible. Visually speaking, this clears up the simulation and creates the characteristic swirls fluids create when being stirred, a phenomenon that can also be observed in Figure 3.2. Mathematically speaking, this stems from the fact that the computed velocity field, after running through all the steps discussed in this chapter, needs to be divergence free cf. [Fer04]. To achieve this, as stated in the Helmholtz-Hodge Decomposition Theorem [Hel], the gradient of the respective pressure field needs to be subtracted from the velocity field. The gradient is calculated from the pressure texture of the simulation by solving the Poisson-pressure equation. This is also described in the Helmholtz-Hodge Decomposition Theorem, the pressure field can be obtained using Equation 3.1. Since the second Navier-Stokes equation enforces $\nabla \cdot u = 0$, this can be simplified to $\nabla^2 p = \nabla w$. Figure 3.9 illustrates the gradient being subtracted from the velocity field. Aside from the clearing up and creating swirls, this

step also makes the fluid mass conserving in accordance with the first Navier-Stokes equation, cf. Equation 2.1.

w ... new fluid velocity (field)

u ... old fluid velocity (field)

p ... density

$$\nabla w = \nabla * (u + \nabla p) = \nabla u + \nabla^2 p \tag{3.1}$$

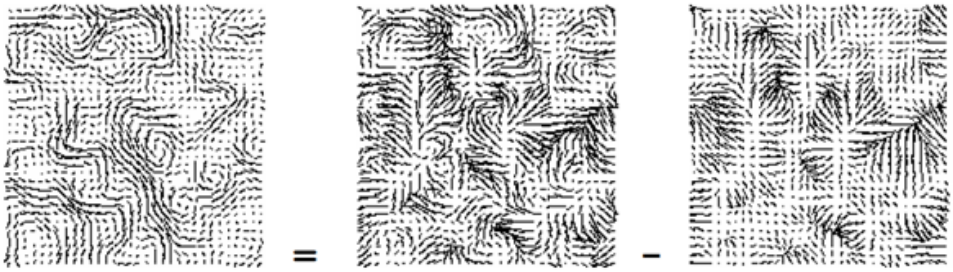


Figure 3.9: The final, incompressible field, illustrated on the left hand side, is obtained by subtracting the pressure gradient, the picture on the right, from the field with divergence, the picture in the middle (Image taken from [Sta03])

CHAPTER 4

Implementation

In this chapter I present the implementation of the method described in Chapter 3. The basis of this thesis' implementation is Stam's work. Given that Stam's paper "Stable Fluids" was published now over 22 years ago, the implementation of the method he describes is by no means up-to-date with the technological possibilities that have arisen in the last two decades.

The biggest change, of course, as mentioned earlier, is that the simulation described in this thesis is GPU-based. The steps discussed in Chapter 3 are mostly implemented in shaders. As is the nature of shaders, they can be executed multiple times in parallel. The steps need to be run through in a specific consecutive order, a step needs to be completed for the entire grid before the next step can be computed.

4.1 Program pipeline

The implementation in this thesis uses two 3D textures (the grids), labelled velocity and pressure texture. Both textures have two buffers attached to them, the front buffer and the back buffer. The back buffer is also referred to as write buffer, while the front buffer is also referred to as read buffer. The two buffers are essential for the parallelism. As the shaders modify values in the grid in parallel, a shader that needs to read a value from neighbouring grid cells and writes a new value to the cell it operates on, could potentially write a different value if the neighbouring cell was or was not changed before the operation. To prevent the shaders getting into each others way, the shaders read from the front buffer and write into the back buffer. After a simulation step is done calculating values, the buffers are flipped. For each simulation step the values the previous step calculated are saved in the front buffer and can be read without disturbance of the write process. This is also why the program pipeline has a very specific consecutive order.

Advection

The illustration 4.2 shows the flow of the simulation. Starting with the advection shader, the shader calculates values reading from the front and writing into the back buffer. After the shader has gone through all grid cells, the buffers are swapped.

Force Addition (Impulse)

If an impulse was given either through user input or a random generation, the impulse shader calculates its values and, again, after the shader is done, the values are swapped.

The droplets approach is relatively straightforward. However, the vector definition requires a more complex approach. A vector can be defined by two points. This is done in the simulation, as illustrated in Figure 3.4 (Renderer -> Input -> Add Force). When a force is added, the respective shader receives the two points and creates a vector from them. The shader then goes through every cell of the volume, adding that vector, multiplied by the inverse relative distance to the start point (If the distance is nought, the multiplier is set to 1). The further away a cell lies from the start point, the smaller the effect of the force. This is illustrated in Figure 4.1 for a 2D example. The 3D version behaves homogenously in all directions, creating a sphere of decreasing influence. The cut-off point of the force is defined in the shader. If an influence is below the determined threshold, it is reduced to nought.

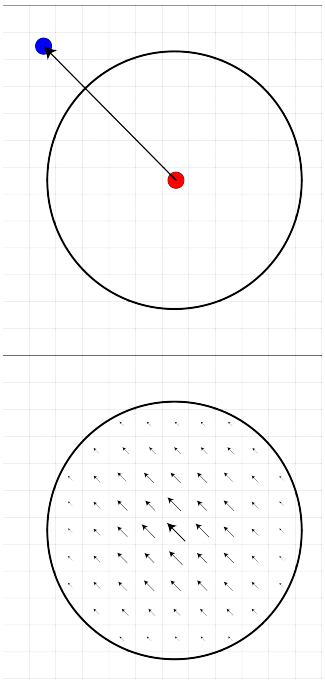


Figure 4.1: Adding a force decreasing in magnitude to each cell in the volume

Diffusion

The diffuse step uses the Poisson Solver to solve the partial differential equation for viscous diffusion (Equation 4.1), more specifically its implicit form as suggested by Stam [Sta99] (Equation 4.2).

$$\frac{\delta u}{\delta t} = v\nabla^2 u \tag{4.1}$$

$$(I - v\delta t\nabla^2)u(x, t + \delta t) = u(x, t) \tag{4.2}$$

The Poisson solver uses a Jacobi Iteration shader as introduced by Golub and Van Loan [GVL96]. The code for the shader, as illustrated in Source Code 4.1, is fairly simple.

```
ivec3 clamp_coord(ivec3 coord, ivec3 size)
{
    return clamp(coord, ivec3(0, 0, 0), size);
}
void main()
{
    /*
    fieldx_r is the old velocity field buffer that values are
    read from (front buffer)

    left, right, top and bottom are the fields connected to
    the cell that a value is being calculated for

    field_out is the buffer that the updated value is
    written to (back buffer)

    alpha and beta are scaling variables, alpha describes the
    relation between grid scale (the scale from grid cells to
    pixels), whereas beta is set to a fraction of alpha
    */
    ivec3 coord = ivec3(gl_GlobalInvocationID);
    vec4 left = imageLoad(fieldx_r, clamp_coord(coord +
    ivec3(-1,0,0), imageSize(fieldx_r)));
    vec4 right = imageLoad(fieldx_r, clamp_coord(coord +
    ivec3(1,0,0), imageSize(fieldx_r)));
    vec4 top = imageLoad(fieldx_r, clamp_coord(coord +
    ivec3(0,1,0), imageSize(fieldx_r)));
    vec4 bottom = imageLoad(fieldx_r, clamp_coord(coord +
    ivec3(0,-1,0), imageSize(fieldx_r)));
    vec4 front = imageLoad(fieldx_r, clamp_coord(coord +
```

```
        ivec3(0,0,-1), imageSize(fieldx_r)));
        vec4 back = imageLoad(fieldx_r, clamp_coord(coord +
        ivec3(0,0,1), imageSize(fieldx_r)));

        vec4 center = imageLoad(fieldb_r, coord);

        vec4 result = (left + right + top + bottom + front +
        back + (alpha * center)) / beta;

        imageStore(field_out, coord, result);
}
```

Source Code 4.1: Jacobi Iteration shader taken from Inkbox [Ink]

The shader is executed multiple times. The default numbers of iterations for the Diffusion and Projection step are 40 and 20 respectively - this is discussed further in Chapter 5. At the end of every iteration, the buffers are swapped.

Boundaries

The boundary shader, as described in the last chapter, sets forces at the volume's edges to nought c.f 3.8. This is achieved with a shader checking if a cell is on the edge and which edge the cell is on. If the cell is an edge cell, the force is set to nought in its respective component.

Projection

The projection step is composed of three smaller steps. As explained in Chapter 3, the goal of the projection step is to get the pressure field's gradient and subtract it from the velocity field to create an incompressible field with observable swirls in the flow cf. Figure 3.9. The first step here is again to solve a Poisson equation. This time the Poisson equation for the pressure field. Again, the step relies on a Jacobi Iteration Solver, however, the divergence is first calculated from the velocity field with a shader as illustrated in Figure 4.2.

```
void main()
{
    /*
    fieldx_r is the old velocity field buffer that values are
    read from (front buffer)

    left, right, top and bottom are the fields connected to
    the cell that a value is being calculated for
```

```
    field_w is the buffer that the updated value is
    written to (back buffer)

    gs is the simulation's grid scale, i.e. the scale used to
    scale between grid cells and pixels
    */
    ivec3 coord = ivec3(gl_GlobalInvocationID);
    vec4 left = imageLoad(field_r, clamp_coord(coord +
    ivec3(-1,0,0), imageSize(field_r)));
    vec4 right = imageLoad(field_r, clamp_coord(coord +
    ivec3(1,0,0), imageSize(field_r)));
    vec4 top = imageLoad(field_r, clamp_coord(coord +
    ivec3(0,1,0), imageSize(field_r)));
    vec4 bottom = imageLoad(field_r, clamp_coord(coord +
    ivec3(0,-1,0), imageSize(field_r)));
    vec4 front = imageLoad(field_r, clamp_coord(coord +
    ivec3(0,0,-1), imageSize(field_r)));
    vec4 back = imageLoad(field_r, clamp_coord(coord +
    ivec3(0,0,1), imageSize(field_r)));

    float div = (right.x - left.x)/(2 * gs) + (top.y -
    bottom.y)/(2 * gs) + (back.z - front.z)/(2 * gs);

    imageStore(field_w, coord, vec4(div, 0, 0, 0));
}
```

Source Code 4.2: Divergence shader taken from Inkbox [Ink] - clamp function cf. Source Code 4.1

After calculating it, the divergence can be used as fieldb_r parameter in the Jacobi Solver (cf. Source Code 4.1) to calculate the pressure field gradient. Once the pressure gradient is determined, it can be subtracted from the velocity field in a subtract shader to create an incompressible field with swirls cf. Figure 3.9.

4.2 Adding particles

Particles can be added to the simulation, to simulate additional fluids being introduced into the system. Much like the addition of forces, adding particles is achieved by choosing a coordinate and adding a single particle to that point. In terms of implementation this means that the buffers for the particles must be reinstantiated, i.e. instead of laboriously changing buffer sizes on the fly, the original buffers are copied into bigger buffers, deleting the original buffers in the process.

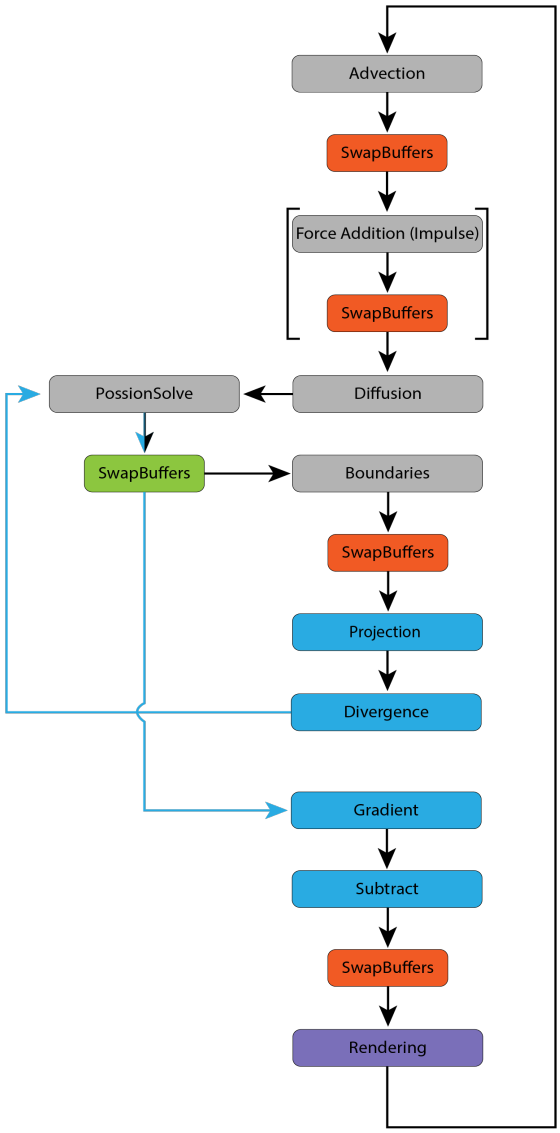


Figure 4.2: Simulation pipeline

4.3 Rendering

Once the final incompressible velocity field is established, the Metaball centres provided by Dynamol can be updated after each visualisation step by simply chaning the buffer Dynamol renders from. This is done using a Transform Feedback shader [Tra]. As with the other shaders, the Transform Feedback uses double buffers to read and write positions. The last simulation time step's positions are read from the front buffer, the respective velocities from the velocity field are added and the result is written back into the back buffer. After this, the buffers are swapped. The buffer swapping here is, again, used to

execute the operations in parallel. Instead of writing data back to the CPU and updating the vertex positions there, this OpenGL feature is used to modify the positions within the Vertex Processing stage. After this step the simulation has run through one time step and starts from the top with the advection step.

CHAPTER 5

Results

In this chapter, I conclude the thesis, analysing the final simulation's performance against the proclaimed goals of this work. Furthermore, I present an outlook on potential future work as well as a description of the challenges this combination of simulation approaches brings in its wake.

5.1 Test environment

All test results were captured in the same test environment. The testing machine is a Dell New XPS 15, cf. Table 5.1.

5.2 Performance

The performance was measured in terms of frame times in milliseconds in the simulation loop. The frame times are summed up in groups of a hundred, averaged, logged and represented in this section. These times, however, are highly dependent on two factors (WorkGroups and Jacobi Iteartions) that can be adjusted in the simulation settings.

Throughout the development of this simulation, I have tested multiple configurations. This section includes recommended settings. Since the testing is based on a machine with specific hardware limitations, however, the recommended settings are primarily relevant for a configuration as itemised in Table 5.1.

GPU	CPU	RAM	OS
RTX 3050 Ti Laptop	Intel i7-11800H @2.30GHz	16GB	Windows 10 Home (10.0.19043 B 19043)

Table 5.1: Hardware Specifications

	Iterations	WorkGroup size	Frame time average	Average fps
Recommended	40|20	8x8x8	51.37s	49.2
More Iterations	80|40	8x8x8	131.49ms	31.3
WG size 4x4x4	40|20	4x4x4	61.36ms	45.4
WG size 1x1x1	40|20	1x1x1	985.71ms	8.0

Table 5.2: Frame time results - "WG" stands for "WorkGroup"

5.2.1 WorkGroups

A major factor in the simulation's performance is the WorkGroup size for calculating the simulation steps. WorkGroups are sets of threads arrangeable in one to three dimensions [Com]. The purpose of WorkGroups is to synchronise calculations for the simulation, simultaneously working on sections of the grid. The size of a WorkGroup determines the size of the pixel volume worked on by a thread.

The most efficient configuration for WorkGroups in this project seems to be, after intensive testing, a WorkGroup size of x = 8, y = 8, z = 8. For more detailed figures on this, refer to Table 5.2.

5.2.2 Jacobi Iterations

Another factor in the simulation's performance is the number of Jacobi Iterations iterating through the Jacobi program used to solve Poisson systems, cf. Source Code 4.1. The number of iterations greatly varies in the Related Work presented in the second chapter of this thesis. For the purposes of this simulation, 40 and 20 Jacobi Iterations for the diffusion and projection step respectively, provide sufficiently visually pleasing results, while also proving not to be too much of a bottleneck for the simulation's performance.

5.2.3 Frame time results

All of the captures documented in Table 5.2 are an average of 30 records of 100 frame times each, i.e. 3000 frame times, captured over roughly one minute of running the program, the last data set "WG size 1x1x1" being the only exception. This version of the simulation is obviously very inefficient. In one minute, only 6 captures of 100 frame times each could be saved. Hence, this figure represents the average of 600 frame times.

WorkGroup size vs. Iterations

As mentioned before, WorkGroup size and Jacobi Iterations are the two factors that influence performance the most. Comparing the results from another test run, it becomes clear that, after going below 4x4x4 for the WorkGroup size, the effect of lowering the WorkGroup size is more detrimental to the frame time average than increasing the iterations cf. Table 5.1.

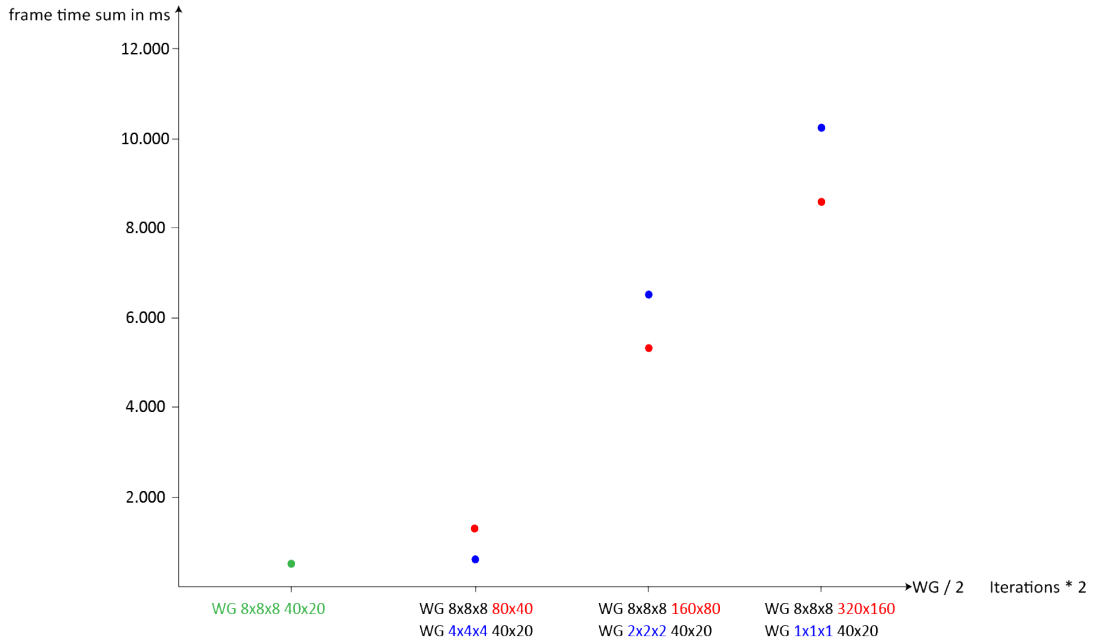


Figure 5.1: Comparing frame time sums for different WorkGroup sizes and numbers of Jacobi Iterations - "WG 8x8x8 80x40" stands for a WorkGroup size of 8x8x8 and 40 and 20 Jacobi Iterations respectively

5.3 Discussion and lessons learned

In the planning phase of this thesis, my advisor, Stefan Bruckner, and I went through different simulation methods. Since the aim of the project was to implement a simulation into the already established visualisation, albeit said visualisation back then only being used to visualise immovable protein structures, we were trying to find a simulation method best fit for combining it with Dynamol's [Bru19] Metaball rendering. After reading through the standard works such as Stam's Stable Fluids [Sta99] as well as Harris' [Fer04] and Stam's [Sta03] subsequent additions to it, we decided to go with a similar approach. Throughout Chapter 3 of this thesis, I present multiple captures of an early 2D simulation created for this thesis, cf. Figure 3.2. The first step in implementing an Eulerian grid simulation like the one Stam describes [Sta99], was to create a basic 2D grid version of the simulation, visualising only the grid velocities themselves rather than using the inbuilt Metaball rendering of Dynamol. The 2D simulation could largely be taken from Inkbox [Ink], with the exception of iteration numbers for Jacobi shaders we had to work out ourselves due to our grid being finer than the one used in Inkbox. The next step, building up a 3D grid from the 2D version, went according to our plan as well. It was only when combining Dynamol's Metaball rendering with the simulation that we realised the implications of working on a Eulerian grid. As mentioned in Chapter 4, the typical swirls observed when stirring especially viscous liquids all but vanished

when visualising using particles moving through the grid instead of visualising the forces directly. This is largely due to the fact that smaller forces are oftentimes ignored in this approach. While in a visualisation of forces even the smallest force is visible, in this approach, particles move according to forces, hence, a strong force moving a particle through a weaker force can overshadow said weaker force as the weaker force may only change the particle's trajectory ever so slightly. Countering this problem required a lot of fine-tuning of simulation variables such as the simulated fluid's viscosity and dissipation. Another issue arose at the boundaries of the simulation volume. Having particles rather than forces but not having a way of directly influencing said particles (cf. Subsection 3.2.4), lead to unexpected behaviour at the simulation volume's boundaries. This issue could not be addressed within the limitations of this thesis but I suggest a possible solution for it in Chapter 3.

5.4 Conclusion and future work

While the simulation produces pleasing results in terms of visuals in certain test cases such as Figure 5.3, there are also some inherent issues with it. Having an Eulerian grid, the simulation cannot rely on particle masses to account for inertia. This would not be a problem, were it not for the visualisation method chosen in combination with the simulation method. Visualising a fluid as a secondary fluid, i.e. a fluid like ink moving in another fluid e.g. water, or even just visualising its forces as a vector force field in an Eulerian grid, as done in the 2D version created for this thesis cf. Figure 3.2, renders visually pleasing results, including beautiful swirls and turbulence. Having particles move according to a velocity field provided by such a simulation, however, obfuscates the visually pleasing effects somewhat, as swirls are not directly visualised but rather just provide forces that affect particles. Furthermore, velocities are individually dissipated in the grid system, however clumping of large amounts of particles cannot be prevented as the particles have no form of collision detection, cf. Figure 5.2. This is less noticeable under perfect conditions, i.e. simulating a flow like in Figure 5.3 with no strong forces and close boundaries particles could clump on, and staying within the limits of these perfect conditions, satisfactory results can be created.

However, this method is much more limited by the aforementioned conditions than necessary. An issue that could be fixed in future work, provided a set of features, such as collision detection, manual swirls and a separate velocity texture and shader to handle boundaries is added. As of now, without these improvements and especially when applying strong forces to the system, the simulation produces less-convincing results than under perfect conditions, cf. Figure 5.4.

Notwithstanding its shortcomings and limitations, I believe that this method approach for fluid simulation provides interesting possibilities to be analysed in future work, as simulating in an Eulerian grid is computationally light-weight and Metaball rendering produces great visual results. Provided a proper solution to handle boundaries is added to the method, this approach could be used in real-world applications.

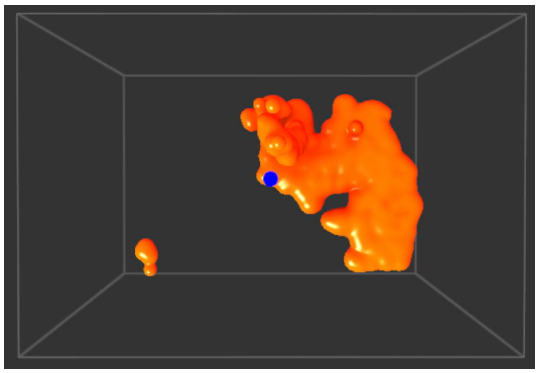


Figure 5.2: Fluid consisting of 750 particles merging at the bottom of the volume (Video [Kon])

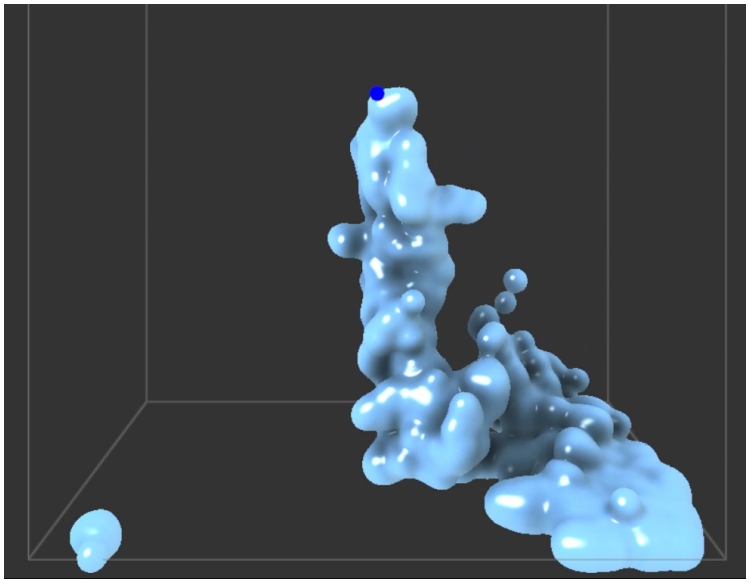


Figure 5.3: Fluid consisting of 750 particles sinking to bottom of volume due to gravity constant (Video[Kon])

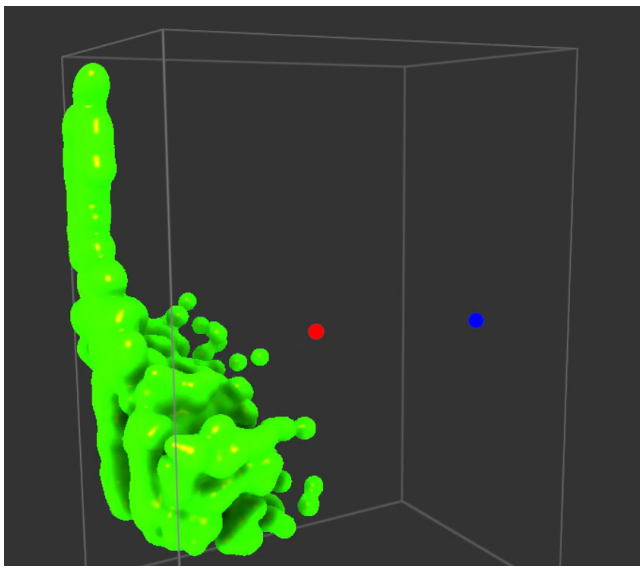


Figure 5.4: Fluid consisting of 750 particles being pushed against a wall not ricocheting from it (Video[Kon])

List of Figures

Bibliography

[AG21] Gonçalo Amador and Abel Gomes. Linear solvers for stable fluids: Gpu vs cpu. In *ACTAS DO 17º encontro português de computaçã gráfica*, pages 1–8. The Eurographics Association, 2021.

[Bli82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982.

[BMF] Robert Bridson and Matthias Müller-Fischer. Fluid simulation for computer animation. In *ACM SIGGRAPH 2007 Courses*. Association for Computing Machinery.

[Bru19] Stefan Bruckner. Dynamic visibility-driven molecular surfaces. *Computer Graphics Forum*, 38(2):317–329, 2019.

[Can88] Claudio Canuto. *Spectral Methods in Fluid Dynamics*. Springer Berlin Heidelberg, 1988.

[cfd] cfdsupport.com. `https://en.wikipedia.org/wiki/Helmholtz_decomposition`. Accessed: 20212-01-05.

[Com] Khronos group documentation compute shader. `https://www.khronos.org/opengl/wiki/Compute_Shader`. Accessed: 2021-12-18.

[DCG98] Mathieu Desbrun and Marie-Paule Cani-Gascuel. Active implicit surface for animation. In *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada*, pages 143–150. the Canadian Human-Computer Communications Society, 1998.

[Fef00] Charles L. Fefferman. Existence and smoothness of the navier-stokes equation. In *Princeton; NJ 08544-1000*, pages 1–7. Ashdin Publishing, 2000.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FM97] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, page 181–188. ACM Press/Addison-Wesley Publishing Co., 1997.

[FS66] Bruce Finlayson and Laurence Edward Scriven. The method of weighted residuals - a review. *Appl. Mech. Rev.*, 19(9):735–748, 1966.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[Hel] Helmholtz-hodge decomposition on wikipedia. `https://www.cfdsupport.com/`. Accessed: 2021-11-26.

[Ink] Inkbox. `https://github.com/bassicali/inkbox`. Accessed: 2021-11-08.

[Kon] Bachelor thesis subpage konstantinlackner.at. `https://konstantinlackner.at/bachelorarbeit`. Accessed: 2021-11-08.

[Luc77] Leon Lucy. A numerical approach to the testing of the fission hypothesis. *Astron. J.*, 82:1013–1024, 1977.

[Met] Ryan's guide to metaballs (aka blobs). `http://www.geisswerks.com/ryan/BLOBS/blobs.html`. Accessed: 2021-11-06.

[MP89] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers Graphics*, 13(3):305–309, 1989.

[Ngu07] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[PAD+17] Runa Nivea Pinto, Asif Afzal, Loyan Vinson D'Souza, Zahid Ansari, and A. D. Mohammed Samee. Computational fluid dynamics in turbomachinery: A review of state of the art. *Archives of Computational Methods in Engineering*, 24(3):467–479, 2017.

[RWT11] Karthik Raveendran, Chris Wojtan, and Greg Turk. Hybrid smoothed particle hydrodynamics. In *Symposium on Computer Animation*, pages 33–42. Association for Computing Machinery, 2011.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[Sta03] Jos Stam. Real-time fluid dynamics for games. 05 2003.

[The] The day after tomorrow. `https://www.imdb.com/title/tt0319262/`. Accessed: 2022-01-04.

[TR04] Nils Thürey and Ulrich Rüde. Free surface lattice-boltzmann fluid simulations with and without level sets. In *Proceedings of the Vision, Modeling, and Visualization Conference*, pages 199–207. Pergamon Press, Inc., 2004.

[Tra] Khronos group documentation transform feedback. https://www.khronos.org/opengl/wiki/Transform_Feedback. Accessed: 2021-12-18.

[Uof] U of t news - and the oscar goes to ... a u of t-educated graphics whiz. https://www.utoronto.ca/news/and-oscar-goes-u-t-educated-graphics-whiz. Accessed: 2021-12-15.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.