Accelerating the Pressure Projection Step in Fluid Simulation Using a Physics Informed CNN

Author:
Madhava Vishnubhotla

First Supervisor:
Alex Telea
Second Supervisor:
Peter Vangorp

Daily Supervisors Traverse Research:
Emilio Laiso

Utrecht University

Faculty of Science
Dpt. of Information and Computing Sciences
Utrecht University
The Netherlands

In collaboration with:
Traverse Research
Breda
The Netherlands

July 26, 2024

Abstract

The rapid increase in ovearll computational power and developement of domain specific hardware has allowed previously offline only techniques to reach the real-time consumer market. Fluid simulations are one such problem that until recently still remained a purely offline endeavor. Extensive research has been done to accelerate the solution of the underlying equations that describe fluid flow. Machine learning methods have recently seen significant success in the realm of computational fluid dynamics. We target pressure projection, the most expensive part of the simulation as a target for optimization/acceleration. By using a CNN we hope to take advantage of highly optimzied libraries in addition to hardware designed specifically to operate efficiently on tensors. Combined with a carefully formulated physics informed loss, the model gains the ability generalize effectively. We build upon work by Tompson et al. By making some informed decisions about model arcthitecture and simplifications in the training process we are able to improve on the current state of the art by achieving faster execution times and higher accuracies across a wide range of scenes. These improvements have brought machine learning based methods closer to being a viable solution for real-time interactive fluid simulations.

Contents

1. Introduction

Fluid simulation is a mainstay in the field of computer graphics. A great many real-world phenomena rely on accurate simulation of fluid flows including everything from water bodies, fire, explosions and clouds with huge variation within each category. In real-time applications, however, the performance budget is minimal and interactivity is of key importance. Even in offline VFX scenarios artist control is desirable, and slow simulations can lead to slower iteration times. Since 1999 and the publishing of stable fluids, real-time interactive simulations of fluid flows has been possible. In the roughly two and a half decades since not much about the base method has changed. However, the requirement for more detail, both in resolution and in the complexity of the flows has increased. Despite computer hardware being much more powerful, the requirement for high quality and high performance algorithms is more important than ever. This thesis aims to explore the various methods developed in recent years to accelerate the simulation of fluid flows and all the associated physical processes. After that a research plan is proposed to systematically compare these classical methods with machine learning based methods to accelerate the most expensive portions of a simulation step.

This master's thesis project was conducted as part of an internship at Traverse Research. The company specializes in research and development of state-of-the-art rendering techniques and high fidelity real-time computer graphics. Their primary focus is on ray traced rendering and consulting with hardware vendors such as AMD, ARM and Samsung. With their recent advancements in volumetric rendering and compression, real-time fluid simulation is a logical next target to showcase their technology. Traverse Research also has extensive experience with machine learning techniques being developed for real-time use. The intersection of machine learning expertise and interest in volumetric simulations led to the foundation of this project.

As machine learning methods have gained prominence in all aspects of computer science, hardware vendors have invested considerable resources into the acceleration of training and inference of ML models. Most popular vendors such as Nvidia, AMD, Qualcomm etc., have begun integrating hardware targeting AI/ML applications. In Nvidia's case for example this manifests as "Tensor Cores" (figure 1.1) that operate alongside traditional floating point and integer arithmetic units. This additional hardware remains unused unless tasks specific to it are given to the GPU.

Coupling real-time simulation with high quality rendering sets tight constraints for the computational budget available for the simulation. Thus, this thesis is an exploration of the experiments and development of an in-house simulator covering potential real-time scenarios

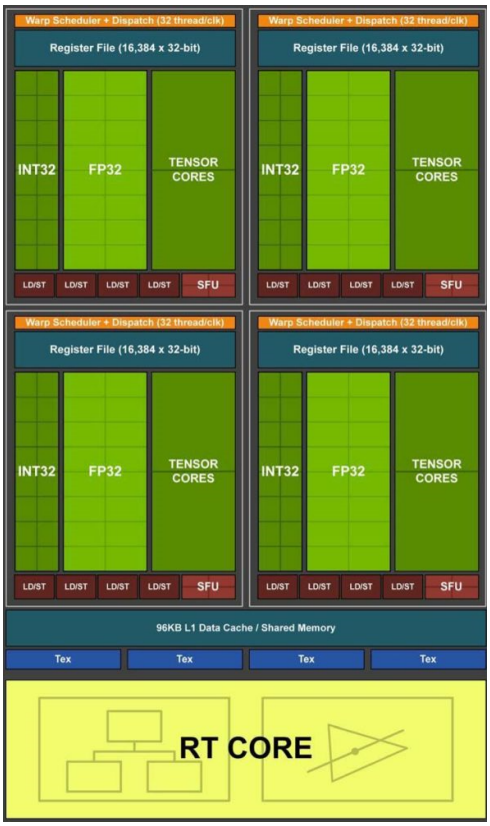


Figure 1.1: Architecture of a modern Nvidia GPU

such as interactive smoke and fire. In order to take advantage of advances in machine learning methods, to accelerate the simulation step and more fully utilize the hardware available to us.

Following this chapter, Chapter 2 gives a broad overview of the history of fluid simulation. The chapter also covers the most popular method called stable fluids used as the basis for the vast majority of fluid simulators used in the games and movie industry.

Chapter 3 takes a broader look at the research done in the field of fluid simulation attempting to capture as much of the broader research landscape in the field of fluid simulation.

We set out a series of requirements in Chapter 4 that define the constraints applied to our project. These constraints are followed by a primary research question along with a sub question that the remainder of the thesis spends answering.

Chapter 5 covers a preliminary experiment conducted to verify the validity of our research question. The result of this experiment is expanded into a broader plan for the rest of the project.

Chapter 6 goes in depth, covering the various components and implementation of a high performance fluid-simulator.

The machine learning portion of the project is explained in Chapter 7, where the previous state of the art is discussed followed by our attempts at improving the architecture. The best models are then fully trained on a complete dataset.

The fully trained models are evaluated in chapter 8 using a few metrics. By analyzing the models' ability to execute quickly, the overall accuracy and ability to generalize is finally used to answer the research question(s) posed in Chapter 4.

In the final chapter (chapter 9), potential directions for future work are suggested. Some of these ideas are potential avenues that this project did not have the time to explore, and others are informed by the broader applicability of the results achieved in this thesis to more complex ideas in fluid simulation.

2. Background on Fluid Simulation

This chapter will take a broad look at the field of fluid simulations, first in developing mathematical intuition and then, more specifically, in the field of computer graphics. As is customary we begin with a brief look at the Navier-Stokes equations. We then take a look at the method published by Jos Stam called Stable Fluids[1]. Subsequently, we will briefly discuss the areas where this method falls short. The next chapter will then discuss the ways in which researchers in fluid simulation and computer graphics have resolved these shortcomings.

2.1 Navier Stokes

The most general form of the Navier-Stokes equations describe the motion of viscous fluids susceptible to compression based on the material properties and their relation to pressure and temperature. These equations have many forms and its properties remain an area of active research.

Those familiar with fluid-simulation in computer graphics might be familiar with a simplified version of these equations known as the incompressible Navier-Stokes equations. As the name suggests the fluids modeled by these equations are assumed to be incompressible. This allows us to simplify the mass conservation problem to a fairly simple condition (equation 2.2). Equation 2.1 is actually 3 equations in one since our velocity has 3 spatial components all of which are also unknowns. In this equation pressure is also an unknown which means that we have 3 equations and four unknowns. The incompressibility condition (equation 2.2) gives us the fourth equation which makes a solution possible.

2.1.1 The Incompressible Navier-Stokes Equations

The method described in section 2.2 solves the following form of the Navier Stokes Equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\nabla p + \mu \nabla^2 \vec{u} + \vec{f} \tag{2.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2.2}$$

These equations consist of two parts: the momentum equation 2.1, and the incompressibility condition 2.2. This is the most common form in which these are written. Our choice for the method of solution (section 2.2) works with this version of the equations. The reader will note

that despite the naming the momentum equation doesn't have the correct units. The rest of this section will also have some simplifications for ease of understanding however a more detailed explanation for this deviation can be found in Bridson's textbook.

$\vec{u}$ is the velocity of our fluid. The first term ($\frac{\partial \vec{u}}{\partial t}$) is how our fluid velocity $\vec{u}$ changes over time. The second term ($\vec{u} \cdot \nabla \vec{u}$) is the self advection term. In a Lagrangian sense, we can think of these equations describing the motion of a single particle or "blob" with a given velocity, through a fluid's flow field. The second term then describes the tendency for that particle to flow along with the rest of the fluid. These two terms constitute the "Material Derivative" or the rate of change of a physical quantity that is placed within a time(the first term) and space (the second term) varying velocity field.

The first term on the right-hand side of equation 2.1 is the pressure term. The negative gradient of the pressure describes the direction in which our velocity should change, i.e. it should move from regions of high pressure to regions of low pressure.

$\mu \nabla^2 \vec{u}$ is the viscosity term. This term makes our fluid particle tend to move at the same rate as the average of its nearby particles. Another way to think of this is that the particle has friction with its neighbors.

The final term is the most general and consists of all other external forces that might influence our fluid's velocity. Some of these forces include, gravity, forces exerted by solid obstacles, wind etc.

Finally, the second equation (2.2) is the incompressibility condition. The simplest way to understand this is that in a velocity field where no additional mass is being added or removed the field has no divergence. A positive divergence would mean that mass is being created, and a negative divergence would mean mass is being removed. One might realize that in many instances this might actually be a desirable property in our fluid simulations, for example when simulating water, we might want a drain and a source of liquid. In our case however we are interested in gaseous simulation, and the most obvious scenario where we might want a divergent velocity field would be in the case of explosions where mass is added to our simulation in the form of combustion byproducts. This is a physical scenario where we aren't necessarily violating incompressibility in our overall flow, but are adding spatially local allowances for deviations in the incompressible condition [2]. This method is discussed in more detail in the next chapter.

2.1.2 Gaseous Incompressibility

In several informal discussions with colleagues and fellow students, when posed the question, "What is the difference between liquids and gases in terms of their flow"?, the general answer was that gases are more compressible than liquids. This intuition isn't completely incorrect,

therefore, we'd like to clarify why in the simulation of gaseous phenomenon we can still make the incompressibility assumption. As mentioned in the previous section, there are cases where this assumption breaks down and must be modified to retain features that are visually interesting.

The first point of clarification is that in our simulations it is not the gas itself that we consider incompressible but rather the flow itself. We make the initial assumption that the air is of uniform density and that the effect of non-air substances in our field do not have the ability to influence the flow directly. In other words they are simply advected by the field and the field, in turn, is not affected by them. As an example, such a case might be one where a dust particles are added to our field. The forces acting on the air will then cause this added dust to move based on the flow of the simulated region of air.

While this example is quite narrow and doesn't cover the range of phenomena we would like to simulate, it becomes and important testing ground for the "correctness" of our simulator. As we will see in section 5.1 an insufficient solution of the pressure equations will lead to visual artifacts in our fluid flow.

The second point comes from the field of computational fluid dynamics (CFD). Here the concept of a "Mach Number" determines whether the flow being studied can be assumed compressible or not. The *Mach Number M* is the ratio of our fluid velocity with the speed of sound:

$$Ma = \frac{v}{a}$$

where Ma is the Mach number, v is the flow velocity and a is the speed of sound in our fluid medium (air in our case). For $Ma \ll 1$ flows can generally be considered incompressible. In some scenarios where we might want to simulate clouds [3], smoke, fire [4] and similar phenomena, we can assume that this holds true (as a simplification only). However, in real-time applications another class of simulation is very common, namely explosions. The sound of explosions comes from the shockwave generated by the combustion which is a supersonic wave of compressed air. This breaks our assumption and more care needs to be taken to handle this case.

The third point of clarification, at the risk of contradicting the previous point, is that in a variety of scenarios the effect of expanding and contracting (compressibility) in gases is, in fact, important. As the reader might recall buoyancy is a force that results due to a difference in density between a fluid and some object/quantity contained within it. In the case of gases the density of a gas is dependent on its temperature. In the simulation of rising smoke the hotter regions should expand, and as other regions cool, contract. This results in the billowy effects we see in real life. We have two choices here, treat buoyancy only as a force that acts on our

velocity field (potentially at the expense of losing some interesting features in our simulation) or also allow for variable density by modifying our incompressibility condition, both of which are discussed in more detail in chapter 5.

2.2 Stable Fluids

Jos Stam's paper Stable Fluids published in 1999 [1] was a turning point in fluid-simulation for computer graphics. The semi-lagrangian (more on this shortly) advection scheme allowed for an unconditionally stable solution of fluid flows. It remains to this day the backbone of many commercial simulators and research projects, including this one. The method consists of a few different steps which essentially boil down to solving each term of the Navier-Stokes equations separately and combining them. Finally, the incompressible condition is enforced which gives us mass conservation. Following is a brief overview of each of the steps. Since we will be modifying and augmenting several of the step described in this paper, it is important to get a good understanding of our starting point.

2.2.1 Eulerian vs Lagrangian Viewpoints

The Eulerian viewpoint considers a given spatial location $\vec{x}$ and observes the change of a given quantity (velocity, fuel, temperature etc.) through that position.

The Lagrangian viewpoint instead considers a 'parcel' which studies the rate of change of its properties as it moves along a fluid's flow field.

This algorithm operates on a uniform Eulerian grid while simultaneously treating grid centers as "particles" as described in section 2.2.4. Subsequent research has implemented solvers influenced by Stable Fluids on other types of grids like non-uniform grids and sparse-grids.

2.2.2 Forces

The paper describes this stage as adding a "source" to our velocity. It essentially boils down to adding any forces our fluid experiences into our field. This term becomes increasingly important once we start simulating fire and smoke since the forces exerted on our fluid, by the combustion and buoyant forces, will be incorporated in this step.

2.2.3 Diffusion

The dynamic viscosity term in the Navier-Stokes equations is solved similarly to how a blur filter works. The velocity of our grid cells becomes the average of the grid cells surrounding it based on a diffusion constant. The paper uses the Gauss-Seidel algorithm which is an iterative method described in more detail in chapter 5 where we compare several solution methods for both this step and the projection step.

2.2.4 Advection

Stam's use of semi-lagrangian advection[5] was the major insight that allowed for unconditional stability in fluid simulations. This method treats each grid center as a particle and traces it backwards in time rather than the standard forward Euler integration we are used to in physics simulation. It is a first order approximation and suffers a lot of numerical dissipation (loss of energy) and diffusion (the spreading out of our velocities away from where they should be). Tracing our velocities backwards in time by doing a backward Euler step is not

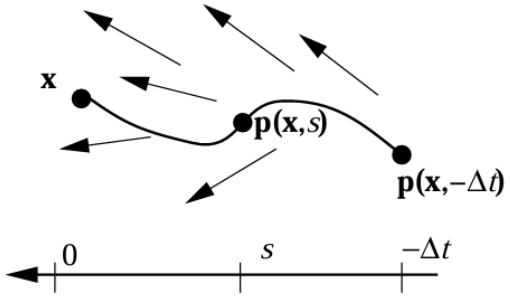


Figure 2.1: Semi Lagrangian Advection: a cell center is traced backwards through time [6]

guaranteed to place us at another grid center. This is countered by linearly interpolating our advected quantity from the four nearest neighbors. Many people have tried to improve this by using higher order interpolators and higher order integration schemes. Since this area has been pointed to as a major source of simulation error we also explore a few of the higher order schemes in this work and figure out the tradeoffs between additional computational cost and quality.

2.2.5 Projection

This step is often described as the most expensive part of the stable fluids solver. It is executed after both the diffusion and advection steps. In order to make our fluid incompressible we want to remove the divergent components of our velocity field. Mathematically this step relies on the Helmholtz-Hodge decomposition which states that every smooth vector field can be described as a sum of a curl-free and a divergence-free vector fields. Intuitively we can see that fluids are swirly which is mathematically described as curl. In order to get our flow field to be divergence free we first calculate the divergence of the intermediate velocity at every point in our grid.

This divergence is then used to solve the non-linear Poisson equation:

$$\nabla^2 p = \nabla \cdot u'$$

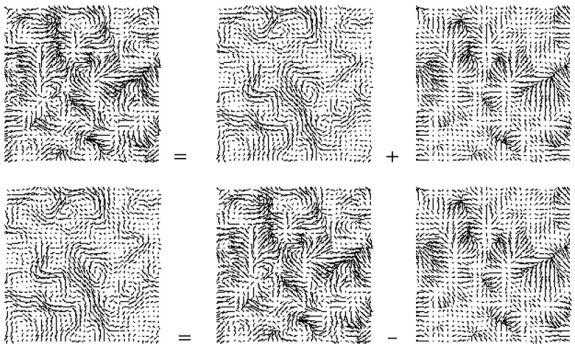


Figure 2.2: Helmholtz-Hodge decomposition [5]

Where u' is our intermediate velocity field after either advection or diffusion. Solving this equation gives us a scalar pressure field. The paper uses the same Gauss-Seidel solver used in the diffusion step.

Finally, the gradient of the pressure computed in the previous sub-step is subtracted from our velocity yielding our final field.

This step has several implications to the quality of our solver. Due to the limited computational budget we cannot simply rely on more iterations to improve the accuracy of our solution. The reason this step is called projection will become clearer in section3.3 where we look a little more closely at another source of dissipation and methods to counteract it.

2.2.6 Boundary Conditions

Readers familiar with partial differential equations will realize that correctly defining the boundary conditions of our problem is incredibly important. Boundaries in fluid simulation consist of two main parts: the borders of our simulation grid, beyond which we do not simulate the fluid flow, and the surface of any obstacle that is placed within our fluid. Correctly setting these boundary values makes a huge difference to the quality and correctness of our solver and needs to be handled carefully.

In our case however, we will not be handling boundaries explicitly and using a procedural system of forces and sources in order to provide users with interactivity and control over the simulation. While obstacles would be nice to have, they can be added on top of the existing system with some extra work. We however want to tackle a slightly simpler problem and gauge its limits in both performance and quality.

2.3 Weaknesses of the Stable Fluids Method

A number of other researchers have developed several methods that reduce the numerical diffusion seen due to both linear interpolation and the single backward Euler step used in the base solver[7]. It is worth comparing these methods and picking the best candidates for our

own implementations and experiments.

In the projection step as we will see in the next chapter there is an inherent loss of energy contained in the divergent component of our intermediate velocity field. Looking into methods developed to counteract this has the potential to reintroduce lost detail in our simulations. One of the oldest methods used to counteract this lost energy is called vorticity confinement[4].

Embergen[8] is the current leading fluid-simulation software built by JangaFx. Their simulator is also based on the stable fluids algorithms. They achieve real-time performance in their simulator and give users the ability to simulate a wide variety of gaseous phenomena. One of the engineers, in a blog post[9], motivates the need for more accurate pressure solves which in turn allows the conservation of mass in our fluid-flow. It is this claim that motivated the first set of experiments we carried out in Chapter 5.

3. Related Work

3.1 Data Structures

The first consideration one must make when developing a fluid simulator is the discretization of the simulation domain. A number of data-structures and memory layouts have been developed to achieve both faster performance and lower memory consumption.

The simplest of these is the uniform grid. The uniform grid spans the simulation domain and each of the volume elements has the same size. There is no preference given to certain regions in terms of computational effort. The simplicity of the uniform grid makes implementing finite-difference based algorithms incredibly straightforward. For higher resolution simulations however, uniform grids fall short for a few reasons. Firstly, wasted computational effort in regions where there aren't any interesting dynamics [10]. Secondly, it tends to have a high memory footprint both due to cubic scaling with respect to resolution and also due to requiring several grids to be stored for each of the phases of the fluid simulation (velocity grid, pressure grid, divergence, curl, density, etc.).

While these are valid issues for the purposes of developing a simulator from scratch it remains the most approachable data-structure [5]. Uniform grids are also an ideal candidate for use on the GPU since they can be represented in memory using 2D and 3D textures. GPU's have hardware specifically built to handle texture operations. There remains one issue however which is that of somewhat incorrect finite differences.

3.1.1 Staggered MAC Grid

Finite differences are a very common way of approximating derivatives and related mathematical operations on a discretized domain. In the experiments described in Chapter 5 the simulator simply uses a uniform grid with pressures, velocities and densities each being stored on the grid centers.

Consider the horizontal direction and the change in velocity over a given grid point i, j. When calculating the partial derivative of the x component of the velocity (u) with respect to $\hat{x}$ we use the following formula:

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \tag{3.1}$$

This is unbiased and accurate as $\Delta x \to 0$. The reader will note that this has the issue of not sampling the grid point at which the derivative is being calculated at all [7]. A forward or backward difference would fix this issue but then would be biased in the forward and backward directions respectively. In order to combat this issue Harlow and Welch **Harlow1965** used a

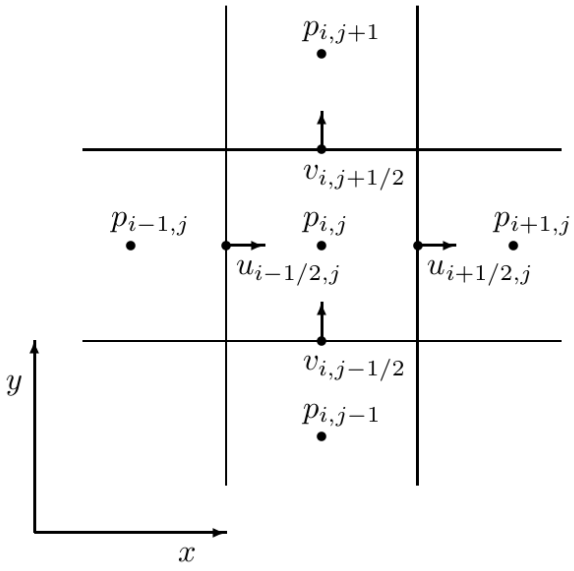


Figure 3.1: Staggered MAC Grid: components of the velocity are stored on cell edges [11] [7]

staggered grid as part of their marker and cell method. This method solves the sampling issue by storing the velocities horizontal components on the vertical walls of our grid cells and vertical components on the horizontal walls. The notation used denotes these as half steps in either direction from our grid point (i,j) in question. The formula thus becomes:

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} \approx \frac{u_{i+\frac{1}{2},j} - u_{i-\frac{1}{2},j}}{\Delta x} \tag{3.2}$$

This is a straightforward extension of the cell-centered uniform grid. As can be seen in the

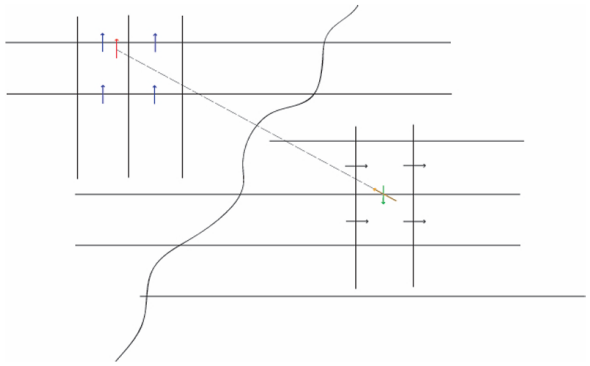


Figure 3.2: Semi Lagrangian Advection on a Stagggered Grid: components traced backwards through time separately **Mbonner**

above figure the routine for semi-lagrangian advection doesn't change much.

3.1.2 Sparse Data Structures

Memory constraints are an important consideration when working with the GPU. This becomes even more important when trying to render scenes with very high quality assets as is becoming increasingly commonplace in both games and media. Pre-simulated volumetric data can take up several gigabytes of storage over the course of a given animation. In most volumetric datasets allocating memory for the entire domain can be wasteful since much of it contains no visual information[1]. While sparse data structures solve much of the memory constraint issues, using them for simulation doesn't always allow us to retain the full speed of a dense uniform grid due to some level of indirection. This tradeoff depends on the size and sparseness of our simulation, of course, but for a wide range of simulations a fixed size domain is sufficient to capture all the details[2].

SP Grid

The Sparse Paged Grid (SP Grid) developed by Setlauri and collaborators [12] is a throughput optimized data structure. Their main goal was to decrease the memory footprint compared to a uniform grid while retaining all the access pattern benefits observed by them. Stencil access is a common operation in fluid simulation. It refers to the situation where a computation refers to a small portion of the total grid such as the diffusion stage where each cell depends on the values stored in its 4 adjacent neighbors. Optimizing for stencil access while also using a

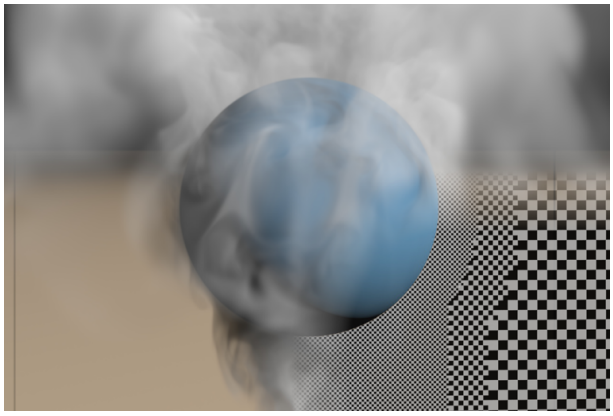


Figure 3.3: SP Grid [12]

virtual address space allowed the authors to achieve a much lower memory footprint. They additionally benchmarked their data structure against a popular and industry standard data structure, the VDB, and densely allocated C style arrays. They noticed faster performance than VDB for both streaming and stencil operations. The primary limitation that this method has is

[1] In a simulation even seemingly empty regions can affect the overall flow. When dealing with sparse data structures we have to carefully account for the regions around our immediate domain. Generally this is handled by allocating an small region beyond our current fluid flow to allow for quantities to be advected into them over the next simulation frame.

[2] Recent work done by JangaFX experiments with movable domains which allows for the simulation region to move along with the area of interest while still using a uniform grid

that it has a slightly smaller maximum address space than VDB and that it does not support out-of-core computations(computation requiring data not stored in physical memory).

Inspired by SP Grid, DC Grid was developed as a GPU specific implementation. The data structure developed here allowed high performance simulation while retaining adaptivity on the GPU [13].

VDB

The VDB data structure [10] is the industry standard for storing volumetric data. It's a multi level tree that allows for fast sequential and random access. It has a theoretically unbounded address space. Due to its ubiquity in the industry many tools and software support it natively. Traverse research also has an internal implementation which allows the reading of VDB files.

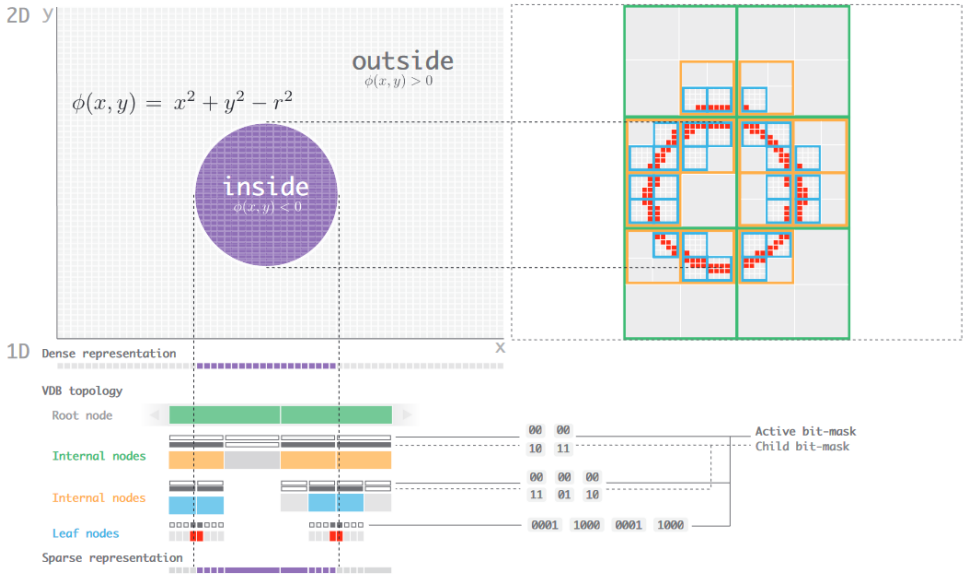


Figure 3.4: VDB: An example of how the data is laid out in the VDB data structure [10]

Brickmap

The brickmap structure is another hierarchical structure akin to the VDB, but instead of 4 levels it only consists of 2 levels. The top level being the "brick map" which maps to the leaf nodes which consist of the bricks. For simulation performance the brickmap has the benefit of fewer levels of indirection compared with the VDB. But compared to a comparable dense grid, for relatively contained domain sizes, it still performs slower. In simulations that occupy small and very dynamic regions, however these data structures are the ideal choice for our solvers. In our case we are choosing to stick to a uniform grid due to ease of implementation.[3].

[3]One might argue that a sparse structure has the benefit of being faster to sample for rendering purposes since large regions can be skipped in the ray marching phase. In our implementation we use a multi-grid method which already requires us to allocate space for several resolutions of our data structure. We can leverage these lower resolution grids to figure out the regions that have minimal/no contribution to our final density grid and skip regions for improved performance.

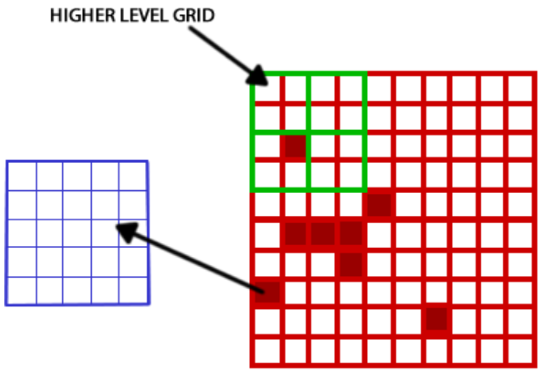


Figure 3.5: Brickmap: The red section shows the dense regions of our domain, for each filled in cell there is more data required to represent it, which is stored in the blue grid (the brick in brickmap) [14]

3.2 Advection

One of the core steps in a fluid simulator is the advection step. The stable semi-lagrangian method introduced by Stam [1] while simple in concept and implementation, suffers from numerical diffusion[15]. This results in a viscous seeming flow, even in the case of inviscid flow. This numerical viscosity isn't artist controllable and in a graphics application should be a tuneable parameter not a byproduct of computational issues. The main cause of this diffusion is the

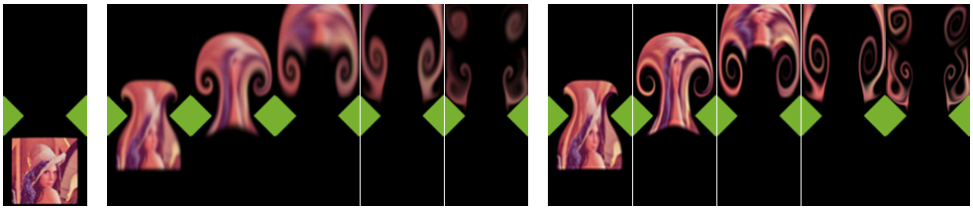


Figure 3.6: BFECC Advection: The left set of six images shows the standard advection compared to the six images on the right which utilizes the BFECC method [15]

linear interpolation. As shown by Bridson a signal being advected using linear interpolation suffers from excessive smoothing and loss of information. This was counteracted by Fedkiw et al. by using a limited Catmull-Rom interpolation [4]. Bridson further suggests a higher order cubic interpolant which results in much less dissipation as seen in the image below. Despite the greatly reduced diffusion the behaviour isn't fully ideal as some smoothing still happens.

Another source of error is the usage of a single backwards Euler step. For applications such as games where we may not always have a fixed time step we run the risk of not following our flow field closely enough. In order to better deal with highly detailed flow we can choose to either take several sub-steps in our advection or use a higher order method such as a backwards runge-kutta 2nd or even 4th order integrator[16]. Each of these are simple to implement and can be a useful parameter that can be changed based on our scene and or computational budget.

Other methods beyond using the standard higher order R-K integrators have been pro-

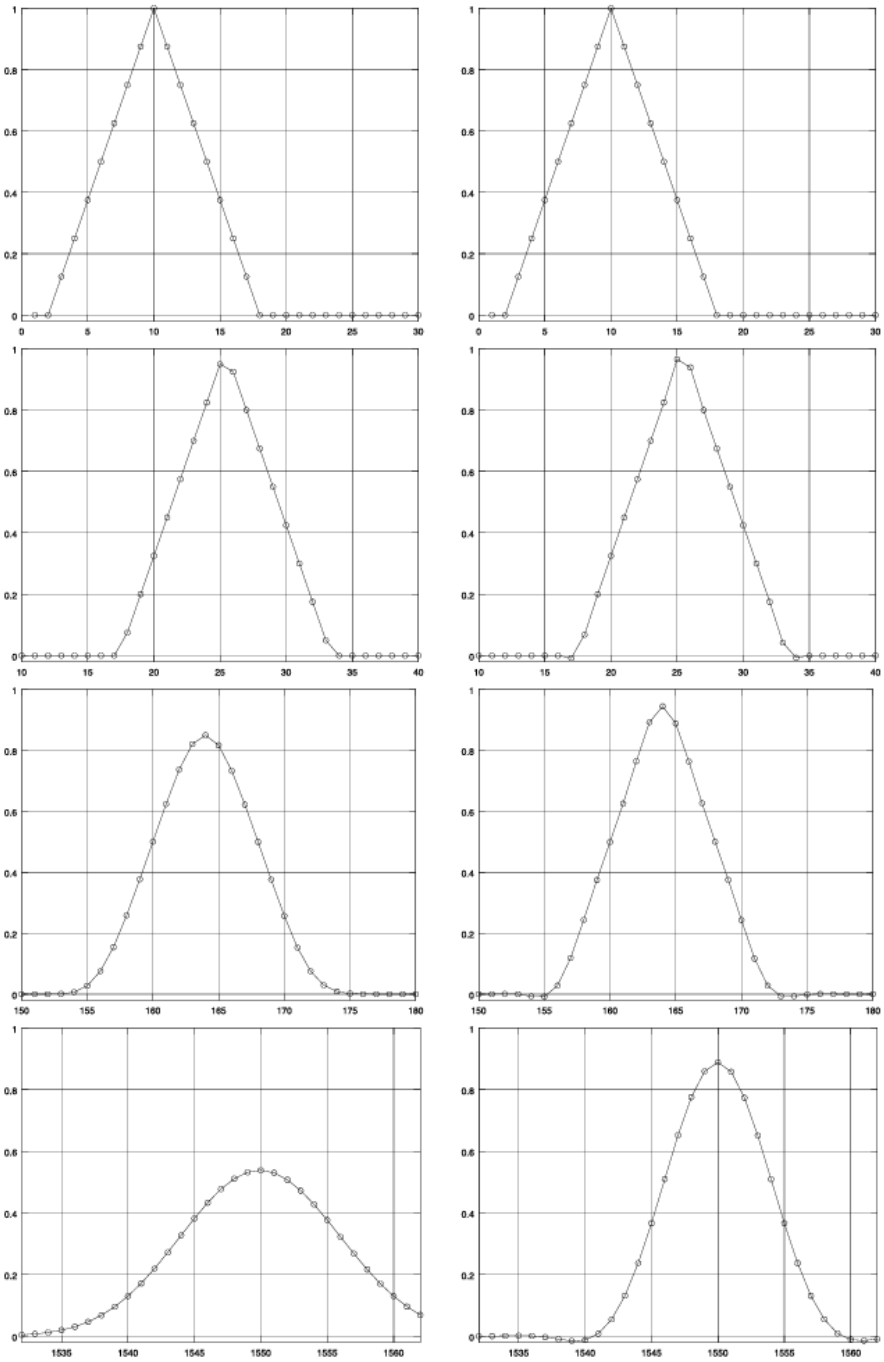


Figure 3.7: Linear vs Cubic interpolation in an advection routine: from top to bottom the left side shows the excessive smoothing caused by linear interpolation compared with cubic interpolation on the right [15]

posed. BFECC [15] or back and forth error compensation and correction works by doing a forward and backward semi-lagrangian advection step and using the discrepancy to create a correction. This gives second order accuracy. The unconditionally stable MacCormack method achieves the same second order accuracy as BFECC while reducing the number of advection steps necessary. [17]

3.3 Projection

The core of our incompressible (and as we will discuss later locally compressible) fluid solver is the projection step. As mentioned in the previous chapter this step is what allows for our flow to be mass conserving. In vector operations we often want to project a vector onto another vector. A vector field analog of this is finding the component of a vector field that goes along another vector field. In our case this is the velocity field after an advection step which can introduce divergence into our flow. By "projecting" our divergent field onto the divergence free component of said field we can enforce the incompressibility condition 2.2.

As we saw in the description of the stable fluids method this step consists of a few sub-steps. The first of which is the divergence calculation (the accuracy of which is improved by using a staggered grid). The pressure calculation. And finally the subtraction of the pressure gradient from our velocity field. The pressure calculation step involves solving a differential equation

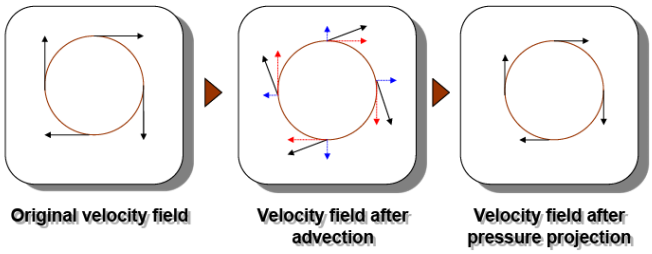


Figure 3.8: Divergent components in the middle (blue) after advection removed due to projection routine (right) causing loss in angular momentum [18]

called a Poisson equation. There are many methods for solving differential equations on a grid some of which we discuss below. This step is the most expensive step in a advection-projection style simulator [19]. Improving the efficiency of this step will be key to achieving high quality fluid-flow in real-time.

3.3.1 Iterative Methods

The equation for solving a given cell's pressure involves the values of its four adjacent neighbors and the divergence of the velocity in that cell. This results in a system of equations which takes the form of a sparse matrix equation. Solving this matrix exactly is infeasible given our real-time constraints, and we must turn to iterative methods to get a "good-enough" solution [20]. Following are a brief description of a number of these iterative methods. Implementation details for each method are further elaborated in chapter 5.

Residuals

In an iterative solver that solves a matrix equation of the form $\mathbf{A}\vec{x} = \vec{b}$ there is an error that gradually decreases with each iteration. In a dynamic application we cannot always calculate

the true error by comparing it with a ground-truth value but would still like to assess how much our solution has converged to our true solution [21]. The residual is the difference between the matrix applied to our estimate minus the right-hand side of our equation:

$$\mathbf{A}\vec{x} - \vec{b} = \vec{r} \tag{3.3}$$

We thus get a value at each grid cell that tells us the imbalance associated with that quantity stored in that grid cell. In our case once the pressure equation is solved and its gradient subtracted from our velocity field, it roughly corresponds to the amount of mass created or destroyed in the projection step. When running an iterative solver applications tend to run the solver until it reaches a preset threshold for our maximum residual:

$$\|r\|_\infty = \max_i |r_i| \tag{3.4}$$

In our case we cannot always afford to run until a sufficient level of convergence is achieved, instead we want to explore which solver gives us the best average convergence over a wide variety of resolutions and scenarios given a specific amount of computation time.

Jacobi Method

The Jacobi method is the simplest iterative solver and for each iteration it updates each grid cell with a new estimate. Its convergence performance leaves a lot to be desired, but its simplicity makes it a good starting point for a first solver.

Gauss Seidel

The Gauss-Seidel method is a simple extension of the Jacobi solver. Instead of only using updated values at the beginning of an iteration it constantly uses the updated values calculated by neighbors. On a GPU this can't be implemented the same way it can on in a serial CPU computation since values that are still being written to will be read from simultaneously. This results in race conditions.

A modification called the red-black Gauss-Seidel method [20] instead splits the computation of a single iteration into two parts where the first half generates a set of estimates and the second pass reads from the first half's estimates. The Gauss-Seidel method therefore doesn't wait as long as the Jacobi method to propagate estimates across the grid and theoretically has twice the convergence performance.

These methods are good at smoothing out high frequency errors quickly but are bad at

reducing lower frequency errors. By weighting our updates with by multiplying it with a weight ω we can over-shoot the estimate leading to faster reduction of lower frequency errors [22]. Choosing the weight correctly is an expensive operation with a similar complexity to solving the exact system of equations. Experiments have shown that depending on the domain size values ranging from $\frac{2}{3}$ for smaller domains to 2 for larger domains work well. This method is known as successive over-relaxation.

Multi-Grid

The multigrid method uses the methods described in the previous section and builds upon them in a clever fashion. The core of a multi-grid solver is the V-Cycle [23]. It consists of the following steps:

- Smoothing: Uses SOR, Gauss-Seidel or Jacobi to smooth out high frequency errors
- Residual Computation: Uses the rearranged Poisson equation to calculate the imbalance in each cell
- Projection[4]: Transfer the values from our fine grid to a half resolution grid, causing our low frequency errors to become higher frequency in this domain.
- Repeat until a coarse enough grid size is reached.
- Now going back up to higher resolutions we interpolate the values back onto our higher resolution grid by applying it as a correction to the existing estimate at that level.
- We perform a set number of smoothing iterations at each level
- This is repeated until the highest resolution grid is reached at which point we stop the V-Cycle

In computer graphics applications, storing multiple resolutions of a texture is known as mip-mapping. While not a one-to-one analogy, the multi-grid method can be thought of as the above operations being propagated up and down a mip-chain [22].

The GPU has a throughput optimized computation model which means that at low resolutions of our grid we aren't fully utilizing the parallel capabilities of our GPU. Nikolay Sakarnykh suggests switching between the CPU and GPU by using a unified memory model like the one made available in the CUDA programming model, based on which processor the current stage of our multi-grid cycle is best suited for [24].

[4]Not to be confused with the projection of the velocity field onto its divergence free component

Conjugate Gradient

The Conjugate Gradient method is a widely used method for the purpose of solving the pressure Poisson equation. The CG method operates like the steepest descent algorithm but instead of following the gradient of our function surface it uses the residuals to guide the optimization. Schewchuk gives an excellent introduction on the method and how it differs from a steepest descent algorithm [25].

While the total convergence speed of the CG method may be faster the closer we are to the solution, when limited to a fewer amount of overall iterations, the setup cost and initially slower rate of convergence on higher resolution grids makes it somewhat infeasible. When we need to achieve higher levels of convergence especially in situations such as liquid simulations [26] where solving the pressure equations as correctly as possible is very important (loss of volume in a liquid is much more visible than in the case of smoke) we can use a preconditioner to greatly speed up the CG method [23].

Similarly to how multigrid methods benefit from a more efficient smoother the conjugate gradient is generally coupled with a preconditioner to accelerate its convergence. A preconditioner provides the initial guess for the conjugate gradient algorithm. The Incomplete Cholesky preconditioner and its variants such as modified incomplete cholesky are commonly used preconditioners. Another option that is becoming increasingly popular is using the multigrid method as a preconditoner for the CG method. Some care needs to be taken in choosing the appropriate projection and interpolation operators to ensure correctness [27]. While multigrid methods are inherently more parallelizable than the IC family of methods, they are less robust to irregular domains and thus more difficult to implement correctly.

3.3.2 Higher Order Projection

Projecting the velocity field onto its divergence free component using the Helmholtz Hodge decomposition removes the energy associated with the curl-free component of the field. This was shown by Zehnder and colleagues where they proved that an Advection-Projection like the one presented in Stable Fluids is only able to preserve energy up to the first order in time . They proposed a solver that would carry out an advection and projection at a half-time step which greatly reduced the energy loss of the system [19]. There is the downside that this requires running the projection operator twice on our field which as we discussed earlier is the most expensive part of our simulator.

Jiang et al. propose an optimization which removes the need for another projection step at the mid-point. They suggest advecting the pressure and pressure gradient to a half-time step and only applying the full projection operator at the end of the full time step. Their method still achieves a second order accuracy with time but at a significantly reduced performance cost [28].

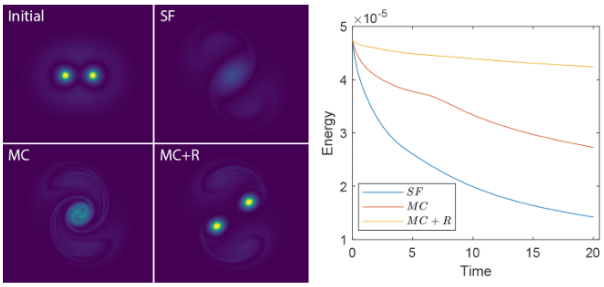


Figure 3.9: Energy Loss due to first order projection in stable fluids vs an advection-reflection solver [19]

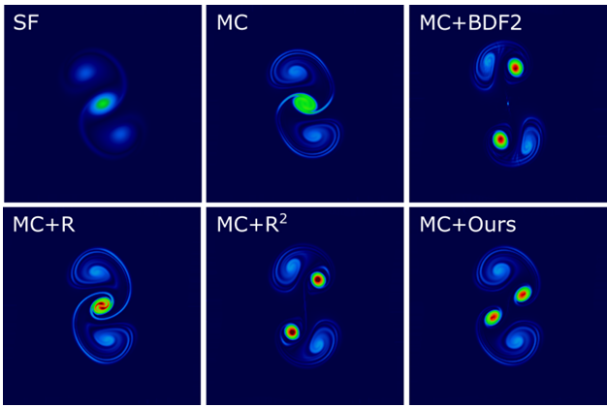


Figure 3.10: Second Order Projection: comparison of Taylor vortices and the detail preserved by several methods [28]

3.4 Detail Enhancement

Calculating fully accurate fluid flow at high resolutions is an incredibly expensive task. Even with the suggested improvements above we may not always be able to achieve the desired level of detail in our simulation. A number of methods have been proposed to reintroduce this detail or upscale a lower resolution simulation.

3.4.1 Animated Curl Noise

Perlin noise is one of the most well-known procedural noises and has been extensively studied in computer graphics. By using perlin noise as a base and making the generated field incompressible Bridson et al. were able to generate plausible flow fields by calculating the curl of a noise. They used this method to create more plausible flow fields that also took into account boundaries and cheaply give the illusion of fluid like motion without carrying out a complete simulation. It is also possible to use this same curl noise as a force field for our fluid simulations to add additional perturbations in our velocity field giving the illusion of more detail.

3.4.2 Vorticity Confinement

Almost every paper includes the addition of Vorticity Confinement as developed by Fedkiw et al. [4]. This method is not only simple conceptually but can cheaply add a great degree

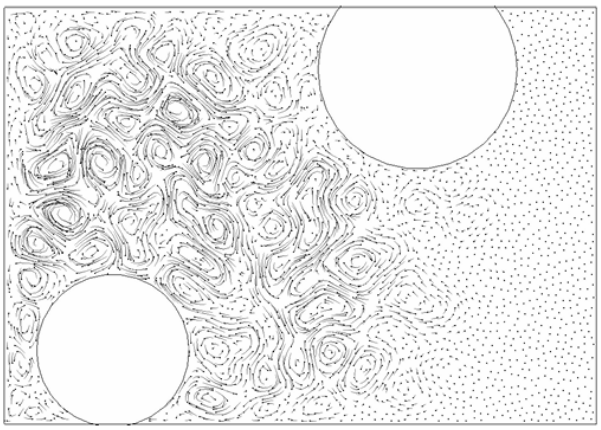


Figure 3.11: Curl Noise [29]

of detail to our simulations. It was this reason that this was implemented as part of the initial experiments carried out in the previous chapter. It operates on the principle that the small scale detail comes from ares with vorticity (curl). Unlike the curl noise method, it uses the flow field itself as a guide for where detail should be added. Then a force proportional to this vorticity is added back in giving us a lot of small scale detail. While successful in introducing detail, the end result suffers from being incredibly high frequency at higher levels of contribution [18].

3.4.3 Vortex Dynamics

Figure 3.12: Vorticity Confinement vs IVOCK: the first three images use vorticity confinement with increasing amounts, compared with the IVOCK scheme which generates a lot more vorticity without sacrificing large scale detail [18]

Vortex Dyanamics provide an alternative way of modelling fluid flow in a lagrangian manner. Instead of modelling the velocity field it models the vorticity of the velocity field instead. Inspired by the methods used in vortex dynamics Zhang and colleagues developed the IVOCK scheme [18]. They observed that the vorticity of the advected velocity field (that no longer preserves incompressibility) should equal the stretched vorticity field given by the advection-stretching function:

$$\frac{D\omega}{Dt} = \omega \cdot \nabla u \tag{3.5}$$

The solution to the above stretching equation is then used to calculate a corrected vorticity field. This field is then used to reproduce the velocity field based on the following equations:

$$\nabla^2 \Psi = -\delta\omega \tag{3.6}$$

$$\delta u = \nabla \times \Psi \tag{3.7}$$

This requires an additional Poisson equation solve along with our pressure calculation. Their method is much cheaper than solving the true velocity-vorticity equations. In addition, they showed that even with much larger time steps, which offsets the additional computational cost, the addition of the IVOCK scheme resulted in improved visual quality overall. The implementation was done on a CPU using multigrid methods, but it might be worth exploring an equivalent implementation on the GPU for real-time use.

3.5 Smoke and Fire

Smoke achieves its distinct look due to the convection caused by temperature currents. In the paper visual simulation of smoke, these effects are modeled as a buoyancy force added to the velocity [4].

$$f_{buoy} = -\alpha\rho\vec{z} + \beta(T - T_{amb})\vec{z}$$

Where $\vec{z}$ is the upwards direction in space. This method does not account for the local expansion and contraction caused by the temperature currents, but achieves visually plausible results.

Simulating fire can be done by adding a scalar field for fuel. If a temperature threshold is reached the fuel can combust and produce fire and smoke. All the fields including temperature, fuel and combustion byproducts are affected by the velocity field and advected along it. Smoke and fire also have additional dissipation terms that are modeled [30]. Its worth noting that despite the Gauss-Seidel method used in the original stable fluids paper, Stam and colleagues opted for a preconditioned conjugate gradient solver for the purposes of simulating smoke and fire. They did not elaborate their reasoning for this choice but as discussed earlier its likely due to the excessive dissipation associated with a less capable solver.

3.6 Explosions

A particularly interesting class of simulations for games and movies is explosions. In games particle systems coupled with sprites of pre-simulated explosions provide the bulk of these

effects. These methods work incredibly well and some degree of interactivity can also be achieved. Simulating them in real-time however has shown promising results with the experimental release of the Niagara Fluids plugin[31] in Unreal Engine[32] which allows artists to couple particle simulations with real-time smoke and fire effects.

By slightly modifying the incompressibility condition (Equation 3.8) Feldman and collaborators [2] showed it was possible to simulate the pressure waves associated with explosions.

$$\nabla \cdot u = \phi \tag{3.8}$$

This doesn't require a complete reformulation of the projection step which is ideal for our use case. They also use a more complicated temperature model as compared with the fire and smoke simulation methods discussed above.

3.7 Data Driven Methods

Physics informed neural networks (PINN's) are an interesting class of machine learning research aiming to incorporate physical laws into the training of neural networks [33]. Using Neural Networks to either replace or augment physical simulations has the potential to accelerate or even improve the evolution of our dynamical systems. PINN's work by extending the loss functions used in the training of the ML models by adding additional terms specific to the physical system being studied. This takes the form of applying restrictions to the derivatives, divergences etc. of our flow fields to enforce the physical laws our simulations should obey.

3.7.1 Upsampling

Despite all the research done into restoring lost energy and reducing numerical diffusion all the methods described earlier are limited by their resolution. Especially for real-time applications it isn't feasible to simply increase the resolution. Several methods have been proposed to achieve more turbulent and detailed results from coarser simulations in CFD. Others have suggested running different components of the solver at different resolutions so that coarser grids can be used for some of our scalar and vector field evolutions and finer grids for the most visually important ones. Upscaling is commonly used in image synthesis and since fluid flow fields are essentially 2D or 3D images, researchers have tried to apply this principle for fluid simulation.

Li and colleagues use this idea of upsampling by doing a lower resolution simulation of the velocity field based on estimated input parameters and then upscaling it for use advecting quantities such as smoke on a higher resolution grid [34]. This reduces the computational cost for a full high quality velocity field calculation by employing CNN's.

3.7.2 Acceleration

Data driven methods for accelerating fluid flow can be broadly put into two categories: predicting fluid flow, and replacing parts of a solver with a learning based method. The former suffers from a lack of generalizability and needing a large amount of training data.

In a paper by Tompson et al. they explored the second method of replacing the projection step in an Eulerian simulator with a convolutional neural network. They developed an unsupervised learning scheme that allowed incorporating information from multiple frames which led to long term stability. This network performed at faster than the typically used Jacobi method for a comparable level of convergence [35].

4. Research Questions

In this chapter we present a detailed set of requirements and constraints set by Traverse Research. We then formulate two questions that we want answered.

4.1 Requirements

Volumetric fluid simulation is an incredibly wide field of study that with numerous classes of phenomena and simulation methods. We want to constrain these to be within the scope a Master's thesis and place hard and soft requirements on the range of effects that we are interested in.

1 Traverse' Breda Framework

The fluid simulator needs to be integrated and developed in such a way that it integrates as seamlessly as possible with Traverse' existing rendering framework Breda. This requires that the simulator be built from scratch using the Rust libraries (crates) developed at traverse, and HLSL compute shaders for all GPU tasks.

2 Eulerian Simulation

Developing a fluid-simulation framework requires a choice between an Eulerian, Lagrangian or a mix of the two. Given the sampling methods used for rendering volumes in the Breda renderer, we constrain ourselves to a purely Eulerian Grid that will contain all of our simulation results. Despite being a semi-Lagrangian method, stable fluids does not track any particles moving through our simulation domain and thus will be our choice for the base solver.

3 Simulation Time

Traverse Research is a company specializing in real-time rendering. This places a hard requirement that all of our computations including the simulation of our fluid, and subsequent rendering all fit roughly within the 16 millisecond time budget of a single frame. This is obviously dependent on the resolution of our simulation which brings us to our next requirement.

4 Resolution

The simulation should be of a reasonably high resolution so as to match the visual quality of a modern game scene. We expect this to be, on average, a domain of size $128 * 128 * 128$ voxels.

5 Smoke and Fire

Some of the most common effects in games are fire and smoke. Being able to interact with these phenomena gives the user an increased sense of immersion. This will be the primary visual effect that we hope to simulate.

4.2 Research question: Efficient machine learning based pressure solver

The partial differential equation describing pressure:

$$\nabla^2 p = \nabla \cdot u'$$

takes the form of a Poisson equation. There exist a large class of solvers that either exactly or approximately provide us with a solution. The exact solutions require the construction of incredibly large sparse matrices, the inversion of which, requires a large amount of memory and computational resources. Therefore, most solvers take an iterative approach that can yield an approximate solution. As discussed in previous sections the Jacobi, Gauss-Seidel, Multigrid and Conjugate gradient methods all belong to the iterative class of solvers. In real time applications however the tradeoff between accuracy and speed is a crucial one. Machine learning methods have shown a lot of promise in scientific applications of fluid flow. Building upon existing work we would like to see how well we can leverage modern machine learning techniques to accelerate the pressure projection step.

Can pressure projection, the most computationally expensive step in a fluid simulation be accelerated to enable real-time simulation of smoke and fire while achieving quality equivalent to or surpassing, commonly used pressure projection solvers such as the Jacobi method?

4.3 Sub Research question: Leveraging domain knowledge effectively

In recent years, the field of computational fluid dynamics and scientific computing more broadly has seen a much higher focus on learning based methods being applied to a wide class of problems. Specifically, Physics Informed Neural Networks (PINNs) utilise a set of ideas that have allowed researchers to more effectively incporporate domain knowledge about the problems being studied directly into the Neural Networks. The ways this is done is quite varied and targets all the stages in a machine learning pipelne. Loss functions can be formulated direclty in terms of the underlying equations of the problem being studied, the architecture can be designed such that it can capture various structural properties (multiscale features), invariants (conservation laws), and even the data can be augmented to capture other variants such as

translational and rotational invariance. It stands to reason therefore, that the pressure projecction step also has properties that can be carefully studied, and by extension incorporated into our machine learning approach.

Can domain knowledge in high performance computing using GPU's and the physics/mathematics of pressure projection be used to make informed decisions about the architecture of our machine learning models?

5. Approach

The first section in this chapter will discuss the setup and results of a preliminary experiment carried out in order to justify that pressure projection is indeed the most expensive part of the simulator and that the quality of the solver has a huge impact on the final perceived quality of the simulation (section 5.1). After a discussion on the findings from the preliminary experiments we developed a step by step implementation plan

5.1 Preliminary Experiments

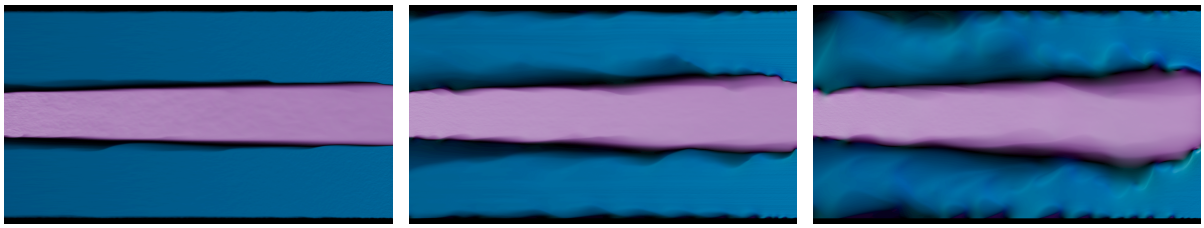
The stable fluids method described in chapter 2 (section 2.2) is fairly simple and numerous implementations exist in a variety of languages. However, getting a strong understanding of each part of the solver is key in order to figure out where improvements can be made. A lot of research has gone into improving the accuracy and performance of the solver. For the purposes of this thesis it is important to figure out the areas that would directly improve the quality of our simulations while maintaining achieving real-time performance (Requirement 3 4.1).

This first version of the simulator was built following the stable fluids solver. It will serve as the building ground on which some preliminary experiments were run. These experiments helped determine the areas that would benefit from optimization and improvement. They also provided some insight into prioritizing the research and implementations for the remainder of the project.

The experiment was set up in 2D as follows:

- Boundary conditions are set up implicitly to allow for free inflow and outflow from any of our four bounding edges. There are no obstacles.
- There is an inflow of velocity from a small section in the left edge and a slightly lower inflow on the right edge but on the entire edge. The inflow on the left also adds in a constant amount of density at each time step for visualization purposes.
- The same experiment was run with 10, 100 and 400 iterations of the Jacobi solver to see the effect it had on the final fluid flow. The resolution of the grid was $2048x1024$. This results in the same amount of voxels as a 128^3 3D domain. While the 3D computation would be more expensive due to an additional dimension of complexity this resolution should provide a very rough estimate of computational cost on a grid size of 128^3.

As can be seen in the figures we notice that at lower iteration counts a considerable amount of



(a) 10 Iterations **(b)** 100 Iterations **(c)** 400 Iterations

Figure 5.1: Effect of increasing Jacobi iterations

detail is lost in our fluid flow and instead of vortices and swirling motion we get 3 columns in our velocity field. The details only start resolving at much higher iteration counts at which point the computational cost becomes very high. 10 iterations at this resolution take roughly 500 microseconds to execute, and at 400 iterations this increases to almost a fifth of a second. Given that not even 400 iterations is enough in this specific instance, we need to greatly optimize the pressure projection step in order to make it viable for real-time use.

As a retrospective on our preliminary experiments we came back at the end of the project to look at what the multigrid solver does when presented with this same scene. The computational cost of the multigrid solution at this resolution is equivalent to 18 Jacobi iterations but the quality and sheer detail (figure 5.2) far surpasses even the 400 iteration solution.

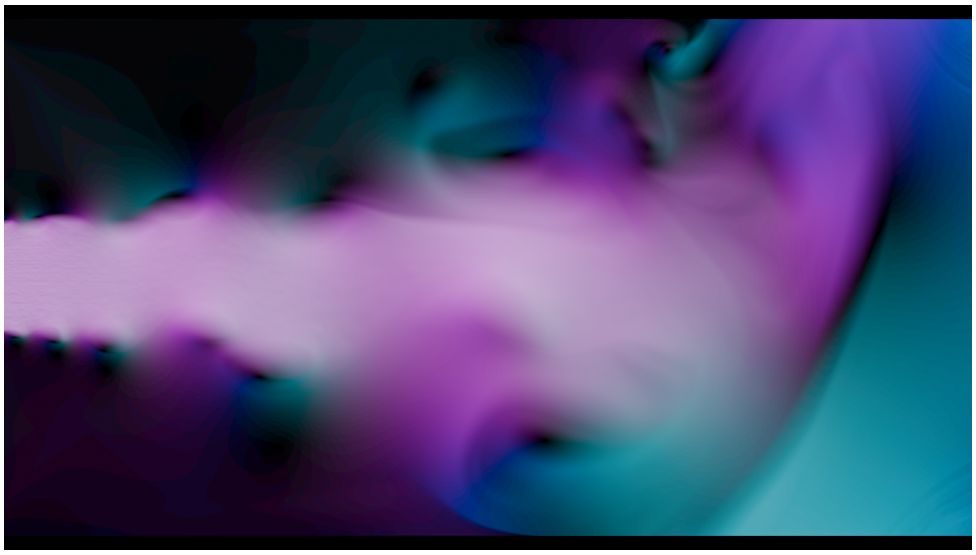


Figure 5.2: Multigrid 1 V-Cycle

The conservation of mass is a fundamental requirement for solving fluid flow. By making our projection step as accurate as possible we can satisfy this requirement. If we can achieve this desired accuracy while reducing the cost of executing 100s of Jacobi iterations it would enable the simulation of highly detailed flows in real-time.

5.2 Towards real-time simulation of fluids

Based on the results provided by the 2D experiments it's clear that a high quality projection is necessary to give us the visual fidelity that modern content pipelines demand. As machine learning becomes more prominent across the computer graphics industry, game engines such as Unreal Engine and Unity have released plugins that enable the use of neural networks within their products. Unreal Engine[32] at the time of writing has released an experimental plugin

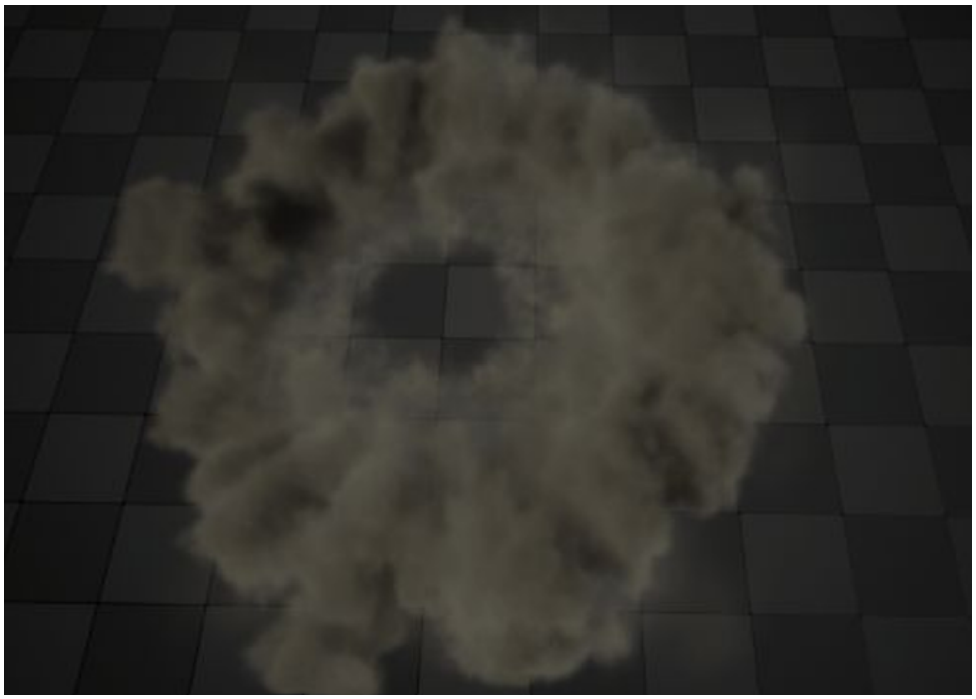


Figure 5.3: Unreal Engine Smoke Simulation

called Niagara fluids[31]. This plugin also uses an iterative solver for the projection step. Exploring their implementation we find that the solver they use is Red-Black Gauss-Seidel with successive over relaxation(SOR). The details of this solver will be explored a little more in the next chapter. Performance wise the Unreal Engine solver, on a Desktop RTX 4070 Ti, takes roughly 7 milliseconds to execute the pressure solve step at a 256^3 resolution. The execution time was estimated by subtracting the frame time at 0 iterations of the solver from 10 iterations of the solver to isolate just the pressure projection step (admittedly, a crude measurement, but it was performed to get a general sense of performance not a strict comparison). Despite being an experimental release, we believe it is reasonable to use their simulator as a baseline target for performance.

In order to do a thorough comparison of fluid simulation techniques we need to build a pipeline from beginning to end that would give us complete control over all of our simulation parameters, data flow etc. Following is a sequence of steps that the pipeline consists of accompanied by a high level overview of each step. More detailed explanations can be found in subsequent chapters.

5.2.1 Adding a 3rd Dimension

The step over from 2D to 3D is straightforward. First we switch from 2D textures to 3D textures. Next we modify the divergence, pressure projection and vorticity confinement formulas to their 3D counterparts. Advection remains largely the same, the shaders simply operate on 3D textures instead of 2D, the semi-lagrangian advection logic remains the same.

5.2.2 Iterative Solvers

The projection step iteratively solves the linearized version of the Poisson PDE:

$$\nabla^2 p = \nabla \cdot \overrightarrow{u'}$$

A number of iterative solvers were implemented, namely, Jacobi, Red Black Gauss-Seidel with SOR and the Multigrid method. Each builds upon the last.

These solvers will serve as the for our machine learning model both in terms of execution speed and accuracy.

5.2.3 A Scalar Accuracy Metric

The primary accuracy metric we use will be the residual norm of the solution provided by our solvers, both learning based and iterative. The lower this scalar value the closer the solution is, to converging, to the exact solution.

The residual for a given matrix equation $Ax = b$ is defined as the difference between the left and right-hand sides of the equation. While not the same as the error, it has the property that it decreases alongside a decrease in error, without requiring a ground truth solution. In the case of the pressure equations the residual equation is:

$$\nabla^2 p - \nabla \cdot \overrightarrow{u} = \overrightarrow{r} \tag{5.1}$$

This gives us a residual value for each cell in our grid, in order to go from this set of values to a single representative scalar value, we can take the norm. We choose the $L\infty$ norm defined as

follows:

$$L^{\infty}Norm = max(|\overrightarrow{r}|) \tag{5.2}$$

5.2.4 Procedural Forces and Sources

In order to have a simulator capable of simulating a variety of scenes/effects we need the ability to influence the simulation in a number of ways. Randomly varying our forces and sources of fuel, temperature, and smoke alongside the simulation parameters we get a powerful tool for generating a considerable variety of scenes quickly.

5.2.5 Data Generation

Using the procedural tools implemented as part of the simulator we generate and save to disk a number of randomly generated scene configurations.

5.2.6 Overfitting Model

To do quick tests of a model's capability we overfit (train on a single scene numerous times) on a simple smoke and fire scene (figure 5.4). This scene is a great candidate for testing our models quickly given we can easily reason about the model outputs and visually compare it with "ground truth" pressure fields generated by our simulator.

We overfit our model on this scene to see if the network has the capacity to learn at least this one scene, and use 10 percent of our frames as validation frames and the remaining 180 frames to train.

5.2.7 Evaluating Model

The validation loss is defined as exactly the L2 norm of the residual(eq 5.3 where N is the number of grid cells/voxels). At this stage looking at the validation loss curve tells us whether it is worth training a given model on the full dataset.

$$L^{2}Norm = \sqrt{(\sum_{1}^{N} |\overrightarrow{r_i}|^2)} \tag{5.3}$$

Figure 5.4: Fire and Smoke

5.2.8 Improving the Model

Building upon the baseline model designed by Tompson et al.[35], we incorporate insights from subsequent research and our own knowledge about the desired structure of the pressure fields. Making informed decisions about the model architecture we aim to simultaneously decrease the size of the model while aiming to increase or retain the overall accuracy of our model.

5.2.9 Train on Full Dataset

Taking into consideration the validation loss of the various model architectures we tested in the overfit stage, we choose the candidates with the lowest losses and train on the full dataset. These fully trained models are then used for our final results and comparisons against the traditional pressure solvers (Jacobi, Gauss-Seidel and Multigrid).

6. Constructing the Simulator

Machine learning frameworks such as Torch, Tensorflow etc., have highly optimized GPU backends. This means that in order to do a fair comparison we need a simulator that also runs on the GPU. Thus, the simulator is built on top of the Breda GPU framework developed by Traverse research. The framework gives us the ability to switch between both Directx12 and Vulkan APIs through a rendergraph API. The API lets us define the resources (textures, buffers, constants) and shader stages, at which point the rendergraph automatically handles the uploading of data and shader dispatch. At this point we can use the bindless system to access all the resources needed by a given shader and define our computations in HLSL shaders.

A custom simulator built in the same framework as Traverse's powerful rendering engine gave us access to a real-time visualizer for our volumetric data, making debugging and designing of the tool much easier. Additionally, being able to save any of our scalar or vector fields at any stage in the simulation gives us a great amount of control in our data generation. The simulator we built, has a powerful procedural system inspired by Embergen[8]. Finally, we utilize a simple GPU profiler that gives us accurate timings on shader execution times, which we use to measure the performance of the various solvers we implemented.

6.1 Simulation resources

Before defining the simulation stages, we need to create the resources for storing our data, and simulation parameters. The primary data structure we use for the simulation is a texture(2D or 3D). We chose textures instead of buffers since most consumer GPU's have hardware support for bilinear and trilinear sampling. GPU drivers additionally have the ability to optimize texture layouts in memory to optimize for spatial locality in each texture dimension. A common mechanism used to do this is called a Morton space-filling curve (figure 6.1). Therefore, when we load or sample a given region in a texture the neighbors are also loaded into the texture cache thereby giving us a modest performance boost. The remaining parameters live in structs that are loaded as needed by each shader stage.

Choosing the texture formats correctly is of crucial importance. Bandwidth is a huge consideration when it comes to managing bottlenecks in our software, and it is self-evident that when loading a 128 byte cache line using a series of 16 bit floats gives us double the "useful data" over a standard 32 bit float. We therefore prefer to use 16 bit floats everywhere we can so long as it doesn't diminish the perceived quality of our simulation. The velocity textures are stored as 4 component RGBA textures, with 16 bits per channel. The last channel is unused

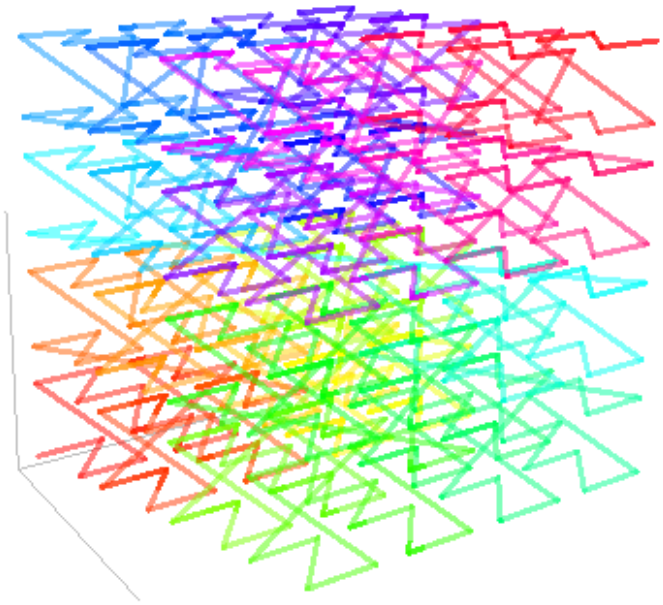


Figure 6.1: Morton (Z-Order) space filling curve

due to alignment requirements where a 3 component texture with 16 bit signed floats each does not exist. The pressure, divergence, density, temperature and fuel textures are all 16bit single channel textures.

In the case of the pressure and divergence textures we found that we hit the limit of 16bit precision fairly quickly when using a multigrid solver and need to resort to a 32 bit texture instead to achieve lower residuals. The 32 bit textures are used only for the generation of training data and in a production environment we would stick with the 16 bit textures. Since our target is real-time applications and not scientific we can draw an arbitrary line regarding what looks "good enough", and in our tests we find that 16 bit precision is almost always "good enough".

6.2 Simulation Stages

The first two subsections describe the procedural controls available to the user. The section following them covers advection which in the actual simulator occurs after projection. Projection being this work's main focus is presented last as it requires the most detail. So the actual simulation loop occurs as follows:

$$\textit{Add Sources} \rightarrow \textit{Add Forces} \rightarrow \textit{Project} \rightarrow \textit{Advect}$$

6.2.1 Adding Sources

Operating in a grid we can define our 3D primitives using signed distance functions (SDFs). SDF's allow us to quickly query whether a given grid cell is within our object or not by checking the sign of the value returned by the SDF[1]. If a grid cell is within our SDF we use 3 floats to determine how much smoke density, temperature and fuel are added to the given cell (scaled of course, by delta time). The source' orientation position and scalar addition rate can be modified at runtime, and animated if connected to an animation system.

We chose a few simple primitives for our sources:

- Sphere
- Box
- Torus
- Cylinder

In a full implementation of this system we would like to generate the SDF's of arbitrary triangle meshes. This is, however, left as a future task.

Once the updated value of the 3 quantities are calculated the same shader handles the combustion stage, which applies cooling due to dissipation, heating and smoke generation due to combustion of fuel. The remaining fuel depends on the burn rate. The final quantities of smoke and fuel that remain are used to add a downward force scaled by their weight, and the temperature is used to add a buoyant force based on the boussinesq approximation[36] (equation 3.5).

6.2.2 Adding Forces

Forces give us the ability to precisely sculpt the overall shape of our flow itself. The gravity and buoyancy forces are applied directly as calculated by the 'add source' (subsection 6.2.1) stage. The procedural forces are then calculated based on vector equations. The choice of forces were inspired by the forces available in Embergen[8].

The force primitives are as follows:

- Line: The line force applies a force in a given direction. The line force has an additional parameter that allows the addition of a tangential force around the line. If a falloff mode is selected then the forces only act in a user specified radius around the line. This allows the user to generate effects like tornados.
- Toroidal: The toroidal force acts around a virtual torus defined by its two radii. The first force acts tangentially to the main circle and a second force that acts tangentially to the

[1] In our case we only need a binary mask that tells us if a cell is within our object or not, and this can be simply precalculated, but at the time of writing is not an optimization we have made.

secondary circle. This gives the user the ability to add extra volume to plumes and create more interesting shapes in the flow.

- Planar: A directional force is applied from a flat circular plane with a given radius. This force allows more precise control than the line force and is akin to a fan, whereas the line force is more like wind.
- Noise: A divergence free noise similar to curl noise, gives the user the ability to add a bit of extra animated detail in situations where the flow is too laminar.
- Vorticity: Vorticity confinement is implemented as described in the paper by Feldman and colleagues. Unlike noise the vorticity force acts based on the existing vortices in the fluid, and serves mostly as a way to restore energy lost due to numerical simulation limitations.

6.2.3 Advection

The advection step is implemented using the semi lagrangian advection scheme (section 2.2.4). During experimentation however it became apparent that the first order backwards advection failed to follow the line force effectively. As researchers have previously pointed out a first order scheme is usually insufficient for most applications and at least a 2nd or 3rd order Runge-Kutta scheme[16] should be used[29]. We implemented up to a fourth order Runge-Kutta scheme which gives improved results in fast moving flows.

(a) 1st Order **(b)** 2nd Order **(c)** 3rd Order **(d)** 4th Order

Figure 6.2: Advection Accuracy

While the number of texture samples scales linearly with the order of the integration scheme chosen, we found that it did not have a significant impact on performance, likely due to GPU's being optimized for texture sampling. Since this operation has to happen only once per simulation step, the number of texture loads is still much lower than in the projection stage. Additionally, we use cell-centered velocities which allows us to perform the backwards integration only once and use the resulting coordinate to sample both the velocity texture and the scalar textures. If a staggered grid were used each component of the velocity would need a separate integration and an additional integration would need to be performed for the scalar fields. Given our goal is performance this was a worthwhile tradeoff as the visual quality of our simulations is still of sufficient quality.

6.2.4 Projection

The projection stage consists of 3 steps.

6.2.4.1 Divergence

The first step is to calculate the divergence of the velocity field. This is done using a central finite difference stencil to calculate the derivatives in each axis of our simulation grid. This corresponds to the following equation, u, v and w corresponding to the X, Y and Z components of our velocity field.

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

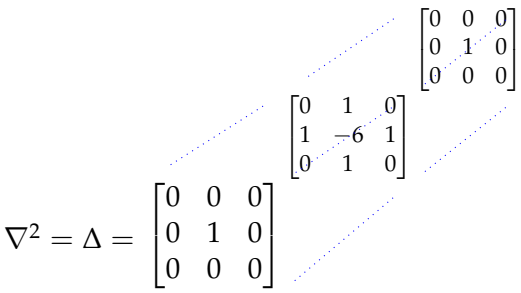
The resulting divergence value is stored in a dedicated texture and loaded into the iterative pressure solver.

6.2.4.2 Pressure Solve

The pressure solver is the core of our fluid simulator. Since the purpose of this work is to find potential speedups in this stage, it is crucial we understand how traditional methods solve it and if we can glean any insights into the structure of the problem that would better inform how we design our CNNs in the next chapter.

We first look at the discrete laplace operator defined as a 3D matrix as follows:

The discrete laplace operator, calculates the second derivative along each axis and can be derived by applying the first order gradient operator described in the previous section twice.



$$\nabla^2 = \Delta = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -6 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 6.3: 3D Laplacian Stencil

The convolution between this operator and the pressure grid gives us the equation:

$$\Delta p = p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6 * p_{i,j,k} \tag{6.1}$$

Substituting this equation into the pressure equation discussed previously we get:

$$\nabla \vec{u} = p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6 * p_{i,j,k} \tag{6.2}$$

Finally rearranging to isolate the center grid cell:

$$\frac{1}{6} * (p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - \nabla \vec{u}) = p_{i,j,k} \tag{6.3}$$

This version of the equation will be true only when the pressure values satisfy the discrete Poisson equation. Therefore, by repeatedly "estimating" the center cell's value based on its neighbours and the input divergence we get closer and closer to the converged value. Another way to think about this is as a blur filter being applied to the grid, but the quantity being blurred isn't the pressure but rather the remaining imbalance pressure from its true values.

The Jacobi Method

The Jacobi method (section 3.3.1)takes the equation and applies it to the entire grid, initializing all pressure values with a 0 at first. It requires two textures, one from which the current estimate is read, and another where the updated estimate is written. By switching between these two textures and repeatedly applying the filter, we get closer and closer to the true solution. Important to note is that it calculates the entire grids estimates, and only then does it swap the texture being read with the texture being written to.

The Gauss-Seidel Method

In a sequential model like the CPU, the Gauss-Seidel (section 3.3.1) method simply loops over each grid cell, and uses the value calculated for the previous grid cell to update the next cell. Since the values are being read sequentially there is no fear of data races. However, if we operate this same concept on the GPU, there is no guarantee that the cell being read from isn't being written to by another thread. To counter this we exploit the symmetry and sparsity of the discrete laplace operator. The laplace operator has 0's in all its corner and edge positions. This means that if we read from a grid in a checkerboard pattern (figure 6.4) and write to the remaining half of the grid, we can avoid data races altogether. This requires that each iteration of the Gauss-Seidel method to be split into two passes, where one half is updated, and then the second half uses these updated values in its update.

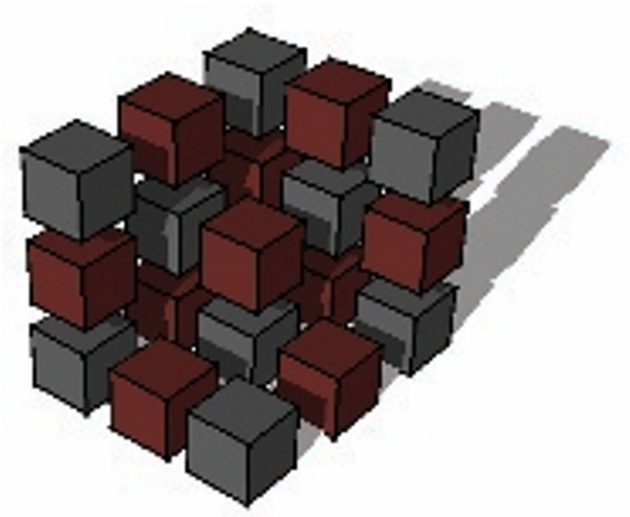


Figure 6.4: 3D Gauss Seidel checkerboard pattern

This method known as the Red-Black Gauss-Seidel solver (red for the first set of checkerboarded cells and black for the remainder) has roughly twice the convergence speed of the Jacobi method since each half pass uses the previous half passes values.

The same texture can be used in both passes and no swapping needs to occur, reducing the memory required. This combined with the improved convergence speed means the Gauss-Seidel method is always a better choice than the Jacobi method. It is only slightly more complicated to implement and is a perfect building block for the next method.

Multigrid

The previous two methods have the useful property of being able to 'smooth' the high frequency imbalances quickly but like with a blur, the larger scale features remain for longer. Rather than blurring at the highest resolution we can downscale our imbalances (the residual) and blur those instead, since each cell at the lower resolution captures more of the overall grid,

causing features that were of a lower frequency at the full resolution to become slightly higher in frequency.

While there are a whole class of Multigrid methods, we chose to implement the Multigrid correction scheme due to its ease of implementation and the ability to reuse other parts of the simulator that need to be implemented anyway. The method was briefly described in section 3.3.1, but we will describe the method with a few more implementation specific details here.

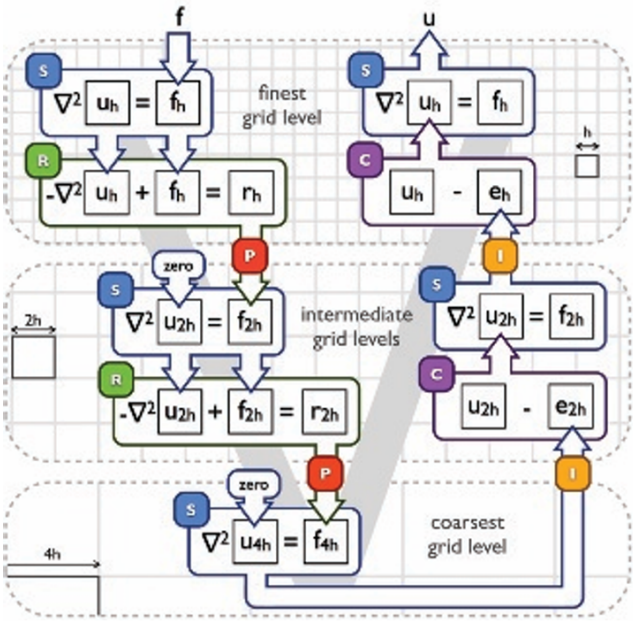


Figure 6.5: 3-level Multigrid V-Cycle[22]

The first step is to allocate the necessary resources for the multigrid method. Unlike the previous two methods we need an additional 3D texture to store the residuals after each smoothing step. We thus have 3 main textures to deal with, the pressure, the divergence and the residual. In this method we treat each texture as an array of 3D textures, where each entry after the main full resolution texture is half the previous texture's resolution. This exactly corresponds to a mip-chain in traditional graphics, however due to current API limitations 3D mips aren't supported across all platforms, and must therefore be manually created and managed.

The steps of the multigrid method are briefly described in section 3.3.1, but a more detailed overview is as follows:

The finest pressure grid (the full resolution grid) is smoothed for a few iterations using the Gauss-Seidel solver. The residual is calculated and stored into the residual texture with the same resolution as the pressure grid (eq 6.2.4.2 where the i subscript represents how coarse the grid is at any given stage in the V-Cycle, 0 for the full resolution grid and increasing until the smallest resolution grid):

$$\overrightarrow{r_i} = \nabla^2 p_i - (\nabla \cdot \overrightarrow{u})_i$$

The residual is then downscaled by averaging sets of 8 cells and stored into the next divergence texture:

$$(\nabla \cdot \overrightarrow{u})_{i+1} = downscale(r_i)$$

The new pressure is estimated by using the downscaled residual as the right-hand side of the Poisson equation instead of the divergence. This lets us operate only on the remaining imbalance from the higher resolution. This process means that each subsequent downscaling operation reduces the problem size by 8 (by 4 in the 2D case, making it an even more powerful solver in the 3D case). The downscaling is repeated until a small enough problem is reached (we arbitrarily chose 8*8*8 = 512). At this each resolution of the pressure field except the highest resolution stores a 'correction' for the imbalance at the next highest resolution (i.e p_{i+1} is a correction for p_i). We simply apply the correction from each cell in the lower resolution (i-1) to the 8 corresponding cells in the next texture (i). But this means that each cell gets the same correction and that means the solution has a few discontinuities. Thus, each upscaling and correcting step, is followed by another set of smoothing iterations. This then smooths out the discontinuities (which manifest as high frequency imbalances after being added uniformly to all 8 cells).

By building a strong intuition of how the multigrid correction scheme works, we are left not only with an excellent solver, but also with a good idea about the multiscale nature of the pressure PDE This will inform our improvements/modifications to the Tompson[35] model in the next chapter.

6.2.4.3 Removing Divergence

After getting as good a solution as we can, we need to use the pressure field to remove the components of our velocity field that are divergent. As described in Chapter 2, this is possible due to Helmholtz-Hodge decomposition of smooth vector fields. By subtracting the gradient of our pressure field from the velocity field we get a divergence free (or as divergence free as possible depending on the accuracy of the pressure solve) field. This is formulated mathematically as:

$$\vec{u} = \vec{u}^* - \nabla p \tag{6.4}$$

Where u* is the velocity field after all the forces have been added to it.

This equation will be our starting point in the next chapter when understanding the unsupervised loss defined by Tompson[35] and collaborators.

7. Machine Learning

With a completed simulator we can move on to data generation and machine learning. We begin our investigation into the best available work done with regards to accelerating eulerian fluid simulation, the model designed by Tompson et al. They employed the use of a convolutional neural network (CNN) to replace the pressure projection stage and saw a performance cost of roughly 34 iterations and the equivalent residual reduction of roughly a 100 iterations of the Jacobi method. The work was published in 2017. Since then a considerable amount of research has been done in the field of Physics and Neural Networks. With our understanding of the pressure projection problem, improved hardware and subsequent research in the field, we worked to improve upon their results.

7.1 The Base CNN

The model takes as input, the divergence of the velocity field and outputs an estimate for the pressure. The loss function calculates the divergence of the velocity after the pressure estimate is used to correct it. Using equation 6.4 we calculate its divergence:

$$\nabla \cdot \vec{u} = \nabla \cdot (\vec{u}^* - \nabla p) \tag{7.1}$$

u^* represents the as yet, uncorrected velocity field. The divergence of this corrected velocity field must be as close to zero as possible. The loss is thus defined as the squared mean of all the values in the resulting divergence field. Rather than directly implementing this loss we observe that by distributing the divergence operator on the RHS we obtain the residual formulation of equation 7.1.

$$\nabla \cdot (\vec{u}^* - \nabla p) = \nabla \cdot \vec{u}^* - \nabla \cdot \nabla p \tag{7.2}$$

$$\nabla \cdot \vec{u}^* - \nabla \cdot \nabla p = \nabla \cdot \vec{u} - \Delta p \tag{7.3}$$

Thus by modifying the loss function we can directly take the divergence fields generated by our simulator which are single component textures as opposed to 3 components for the velocity and directly use it in both our inputs to the model, and also in the loss function.

This greatly reduces the amount of data that needs to be generated saving us time in the data generation step. Training also becomes faster, as less data needs to be loaded at any given time and the loss function is significantly cheaper to evaluate. Physics informed neural networks, and the model we are recreating typically use autodifferentiation to calculate the dderivatives necessary for their respective loss functions. We chose instead to simply run a convolution using the laplacian stencil (figure 6.3). This directly gives us the second derivatives of our pressure without needing to run autodifferentiation twice over our output.

The architecture of this model (figure 7.1) relies on a series of convolution layers followed by a ReLU activation function. The ReLU activation function can only produce positive outputs. If one were to consider pressure physically, there is no such thing as negative pressure, only a negative pressure gradient. However a quirk of the pressure projection method does in fact cause some pressure values to show up as negative (since the input to the pressure solvers can contain negative divergence values). This is as a result of the velocity field having divergences that aren't caused by the physical phenomena of compression or expansion. Thus if we were to compare the output of this network with pressure fields generated by our multigrid solver for example we see that there isn't much correspondance. In practice this does not matter since ultimately we only care about the gradient of the pressure which in the discrete case is just the relative difference between the adjacent grid cells. As long as the gradient is correct then the pressure field generated by this model is also correct.

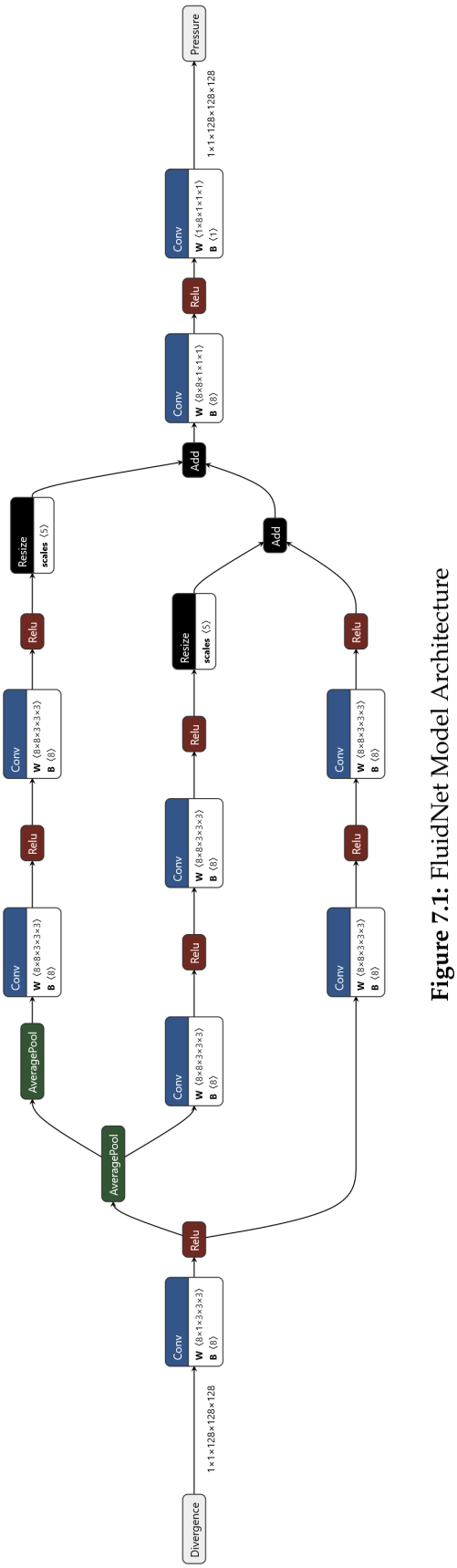


Figure 7.1: FluidNet Model Architecture

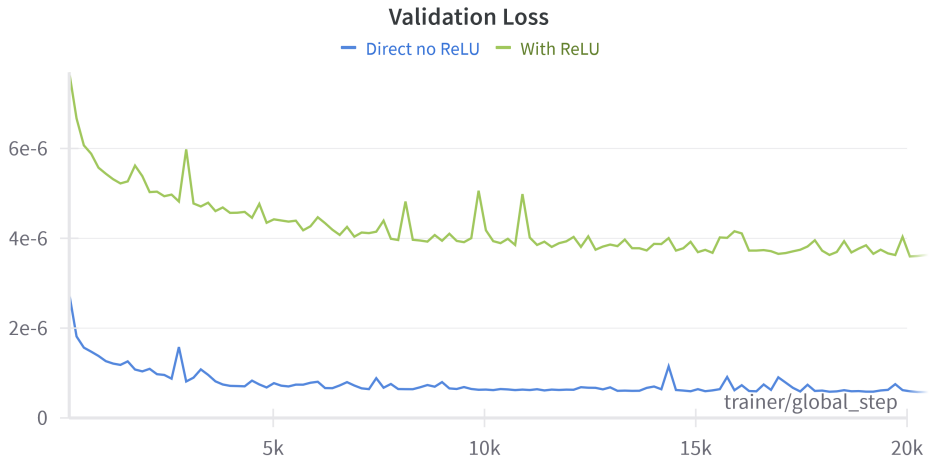


Figure 7.2: ReLU vs no ReLU after final convolution

The only change we make to the model for our tests, is removing the ReLU at the very end (this results in lowered validation loss using the residual loss as seen in figure 7.2)thus allowing the model output pressure fields that contain negative values. This allows the pressure fields from our solvers to be compared directly against the model outputs, rather than the gradient of the pressure fields. This makes it easier to do a structural comparison.

7.2 Evaluating the Overfit Accuracy

With the model and loss function in place, we run an overfit test using a simple smoke plume scene (figure 5.4). The divergence fields were generated using our simulator and 90 percent of the frames were used as training data and the remaining as validation.

We additionally tested a Mean Squared Error Loss using 'ground truth' pressure fields generated by running our multigrid solver on each frame for 7 V-Cycles, but saw worse loss curves compared to the physics based residual loss. This was what led us to realizing we needed to remove the ReLU in order to get visually matching pressure fields.

After training we ran the model over the same 190 frames of the plume scene. The average maximum residual over each frame was in the order of $1 * 10^0$, roughly the same as around 10 Jacobi iterations.

The model's inference time can be measured already at this stage and we can determine whether the performance compares to our hardcoded solvers. Pytorch includes a benchmark module, which gives us accurate timings for a GPU based inference. Using this tool, we find that the model takes 8 milliseconds per inference. Pytorch uses CuDNN as its backend, which uses highly optimized kernels to perform the convolutions at each layer. Additionally, by enabling a flag in our script we can allow the cuDNN backend to choose test several implemen-

tations of their convolution kernels and choose the one that performs the best on our specific hardware. After enabling this flag, the model's execution time goes down to 7.2 milliseconds, a modest improvement.

Given the hardware improvements since 2017 this brings the model's performance much closer to the realm of real-time simulation. By switching to 16bit floating point accuracy (the internal standard used at Traverse) the execution time roughly halves.

7.3 Physics Informed Neural Networks

The work by Tompson et al.[35] was released in 2017, 2 years before the first major physics informed neural network paper. The unsupervised loss however is a great example of incorporating the equations of pressure projection directly into the learning problem. The architecture is also multiscale, which seeks to target the different frequencies seen in the pressure field's features.

Other network architectures have since gained prominence, namely the U-Net architecture, and graph neural networks (GNNs). In a paper by Weymouth[37], the learning is limited to determining a better smoother for use in a multigrid solver through learning based methods. We chose not to mimic this approach as it would have required the simulator and model architecture be very tightly coupled. It would have required considerable engineering effort to either incorporate training of 3D CNN's into the Breda framework, or reimplementing a simulator in python.

Taking inspiration from our understanding of the multigrid solver we propose a few modifications to the base model to more closely mimic the multigrid solver (figure 7.3). Firstly we changed the stages at which the downscaling happens to more closely follow a U-Net architecture, where a convolution first occurs at each level of our network before being downscaled. Another convolution follows the downscaling, and only then upscaled back to the higher resolution grid. Instead of concatenating our upscaled grids like in a U-Net architecture, we directly subtract from the higher resolution grid.

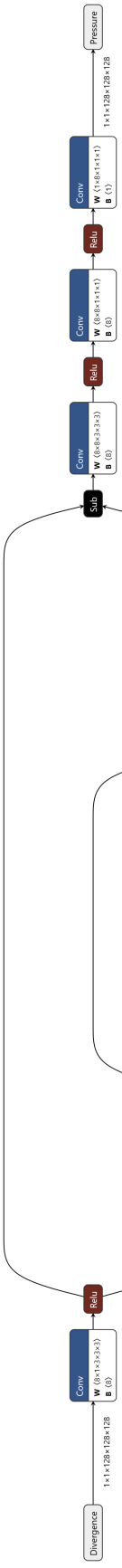


Figure 7.3: MGML Architecture (Our Model)

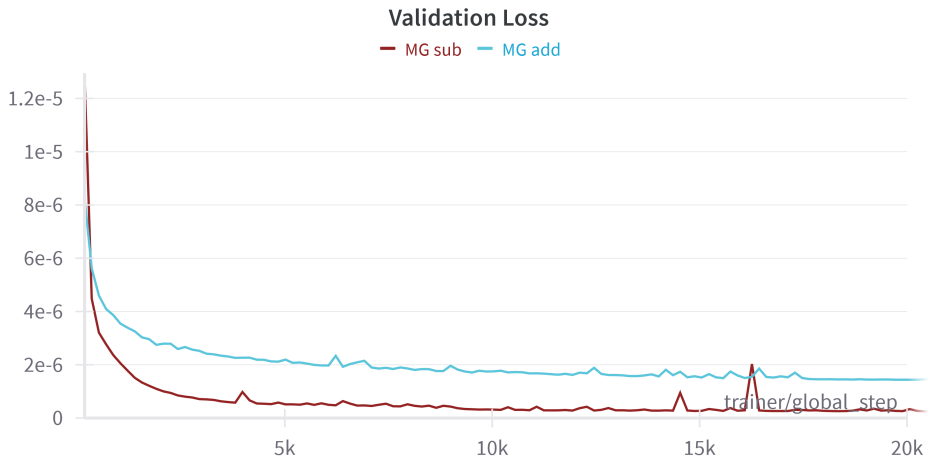


Figure 7.4: Addition after activation vs subtraction

We first tried addition like the base model does, but had worse overall loss curves compared to the subtraction (figure 7.4). An educated guess can be made that due to the ReLU activation functions following the convolutions, the outputs are purely positive, these positive values can only be increased if we add other positive values from other ReLU outputs. In order to give the network the ability to reduce these values as well, we chose to use a subtraction instead of an addition. We also tested variations on this architecture by using a tanh activation function, but again saw worse loss curves leading us to abandon this model. By closely following

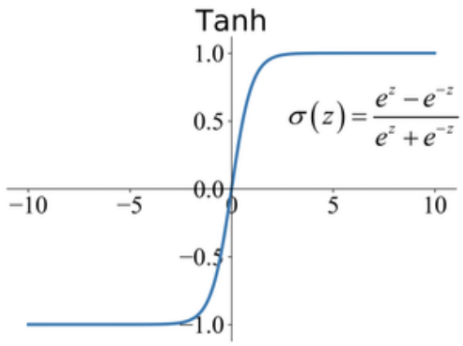


Figure 7.5: tanh activation function

the structure of the multigrid solver, known to be optimal in terms of algorithmic complexity (O(N)) we see improved accuracy over the starting model. Our model has the same number of parameters as the original (3.8k) but faster execution times due to fewer total convolutions performed, but has an average maximum residual over the same test scene in the order of $1 * 10^-2$

This is a considerable improvement by simply rearranging where certain operations occur. A final modification was made to the architecture by having each convolution only generate 4 feature vectors instead of the 8 in the previous two models. This results in a model with only 1k parameters and an 16 bit inference time of just 1.5 milliseconds! The accuracy suffers compared

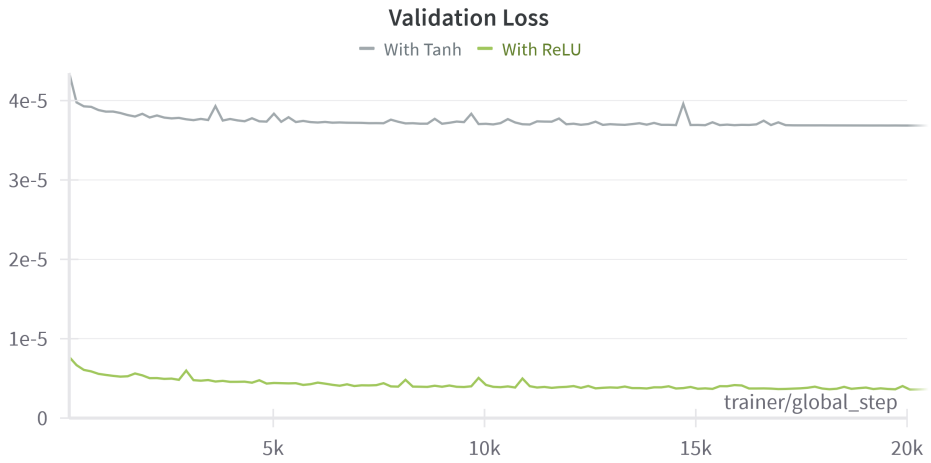


Figure 7.6: tanh vs ReLU activations

with our multigrid inspired model but is still within the order of $1 * 10^{-}1$.

It is important to note at this stage, that all of these values are calculated on a very simple scene, which does not remotely capture all the different effects we wish to capture in a more interesting fluid flow. In the next section we take our best models (figure 7.10). These models are the full multigrid inspired model, MGML, with the same theoretical "capacity" as the Tompson model[35], our reduced size model where we simply reduce the number of intermediate grids generated by each 3D convolution from 8 down to 4, to see how well they generalize given more training data, the Tompson model, and a reduced size version of it akin to our reduced size model.

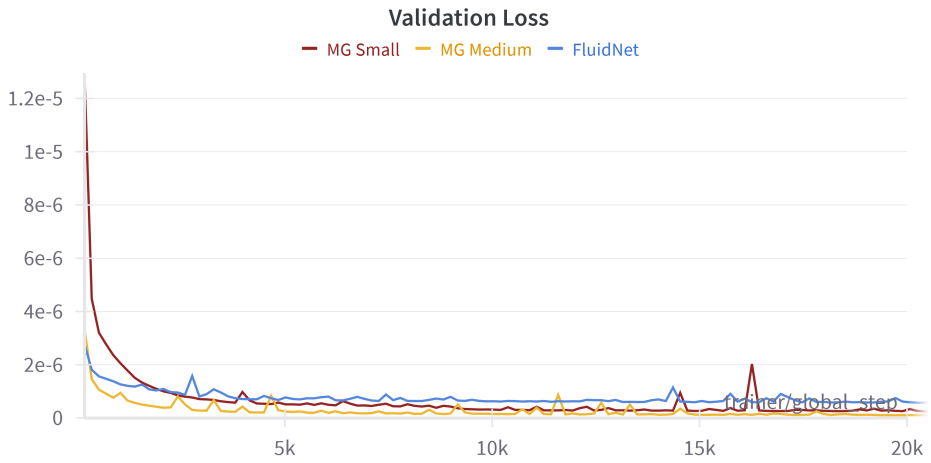
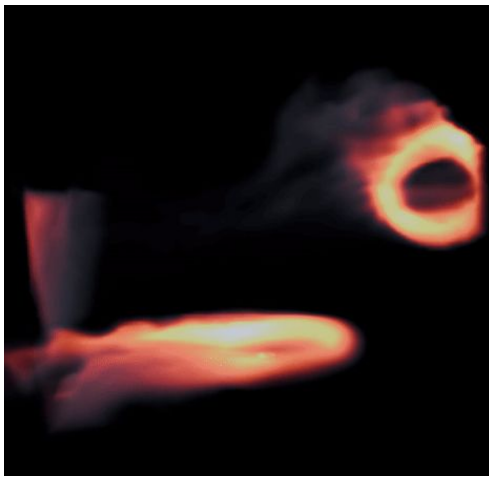


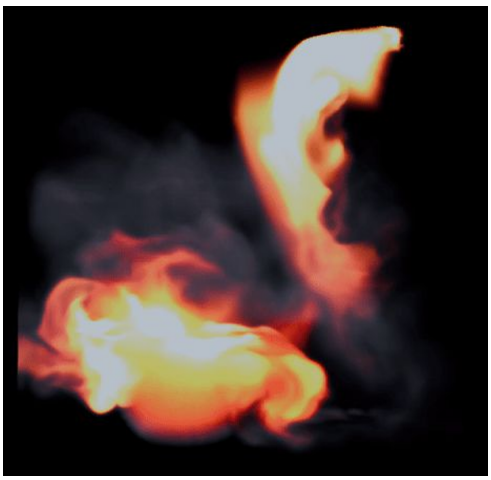
Figure 7.7: MGML vs MGML (small) vs FluidNet

7.4 Fully Training the models

Using our simulator and the procedural system of forces and sources, we created a randomizer that randomly places forces and sources in different orientations and strengths in the simula-

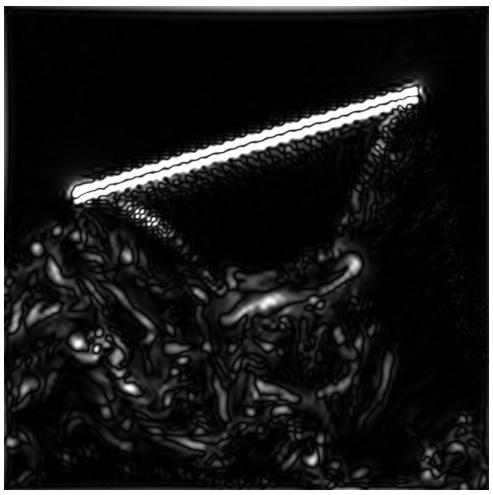


(a) Random Scene 1



(b) Random Scene 2

Figure 7.8: 2 Random Scenes generated by our fluid simulator



(a)

(b)

Figure 7.9: 2D slices of 3D divergence frames from scene in figure 7.9b

tion grid. The randomizer additionally picks gravity, buoyancy, burn rate, cooling rate, etc. at random. Using this randomizer we generate 100 frames of divergence over 200 different scenes (figure 7.9). Divergence frames from 100 scenes were used as training data and the remaining 100 as validation data. We trained each model for 40 epochs (1 epoch corresponds to one pass over the entire 10,000 frame training dataset). The resulting loss curves are as follows:

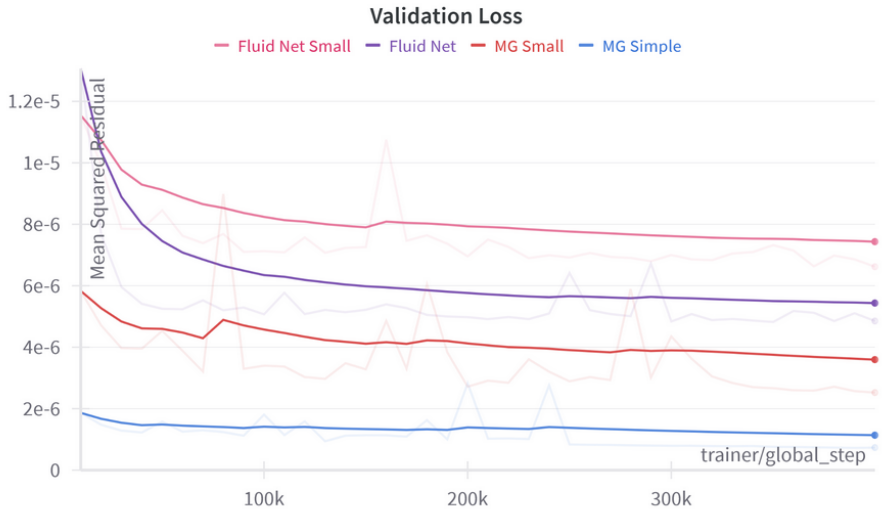


Figure 7.10: MGML vs MGML (small) vs FluidNet vs FluidNet (small) validation loss on full set of scenes

The MGML model "trained the fastest and produces the flattest loss curve, possibly due to the model architecture being the most suited for this specific task. Given that all models managed to train successfully (seen by the relatively flat tails on the loss curves), we can now do a full comparison and analysis of these models' capabilities.

8. Results and Conclusions

In this chapter we go over the final results of this thesis and provide a discussion and analysis on a few main criteria. The first criterion is the raw execution time and the scaling behaviour with respect to input size (section 8.1). The next criterion is the numerical and structural comparison on a variety of scenes (section 8.2).

All experiments were performed on a desktop PC with an Nvidia RTX 4070Ti GPU, 32-core i9-14900k CPU, with 64GB of RAM. The 4070Ti is a relatively high-end GPU at the time of writing and includes acceleration units (Tensor Cores in Nvidia's terminology) for machine learning workloads. Additionally, the performance of all the machine learning models was measured using the highly optimized cuDNN backend provided by the Pytorch machine learning framework. The performance measurements on the Jacobi, Gauss-Seidel and MultiGrid methods was performed utilising the puffin profiler developed by Embark Studios, to get the total compute shader execution times[1].

8.1 Execution Time and Scaling

Tuning solver Performance

There are two main things we want to consider when comparing model and solver performance. The first is the raw inference time at a 128^3 grid resolution as it is the target resolution we consider in our requirements (4.1). The Jacobi, Gauss-Seidel and Multigrid solvers were all tuned so as to match the execution time with the fastest machine learning model (MGML Small).

Table 8.1: Solver Iteration Counts

Jacobi	42 iterations
Gauss-Seidel	34 iterations
MultiGrid	2 V-Cycles

In table 8.1 we can see the resulting iteration counts. Throughout the remainder of the experiments and comparisons the iteration counts are not modified. In table 8.2 we can see the actual execution time with these iteration counts. MultiGrid was given roughly a hundred microseconds less overall in order to, ever so slightly, bias the results against it. The Jacobi and

[1] Its worth noting that the implementations of these solvers is relatively naive and does **not** make use of any GPU optimization techniques such as preloading portions of the grid into shared memory for each thread group, which could improve texture load times.

Gauss-Seidel were tuned to match as close as possible to the MGML Small model.

Execution Time Comparisons

In order to test the overall scaling behaviour of all the different methods we fixed the iteration counts as above, and performed tests at increasing resolutions. By dividing the execution time at a grid resolution of 256^3 with the execution time at a resolution of 128^3 we get an overall idea about the scaling behaviour of the different methods. An exactly linear scaling would demonstrate a scaling factor of 8.

Table 8.2: Scaling Factors

	MultiGrid	Jacobi	Gauss Seidel	FluidNet	FluidNet (small)	MGML	MGML (small)
128^3 grid cells	1.24 ms	1.35 ms	1.33 ms	3.52 ms	2.00 ms	2.26 ms	1.34 ms
256^3 grid cells	6.09 ms	10.24 ms	9.39 ms	34.84 ms	21.47 ms	23.82 ms	14.61 ms
Scaling Factor	4.91	7.59	7.06	9.90	10.735	10.54	10.90

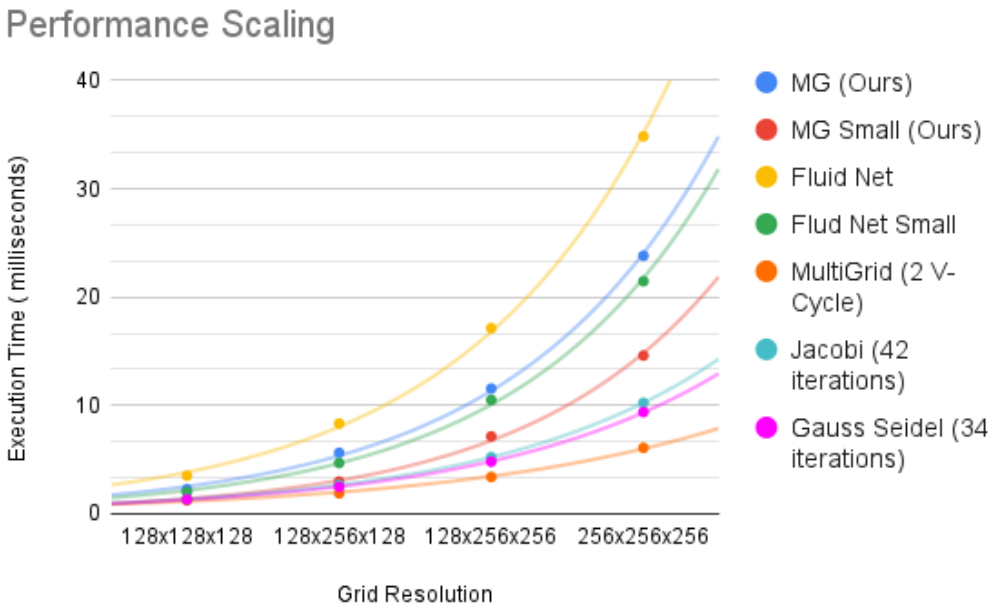


Figure 8.1: Execution Time Scaling Trend

When we consider the scaling behaviour (figure 8.1) we see that the MultiGrid method far outperforms all other methods. If we consider only the machine learning methods then we see that our models execute quite a bit faster than their FluidNet counterparts. Since our total frame budget at 60FPS is just 16.66 milliseconds we need to shave off every millisecond we can. The MGML small is the fastest executing model at just 1.34 milliseconds (table 8.2). Even our full size MGML model with the same total number of trainable parameters as the FluidNet model, is nearly as fast as the reduced size FluidNet model.

8.2 Solver Qualtiy

We consider four scenes for our analysis. The first scene being a randomly generated scene like the ones from the training and validation sets, generated using a distinct random seed. The next scene consists of a simple fire with no additional forces. These first two scenes are just general test cases at a grid resolution of 128^3 where the former scene is generated in a manner similar to the training data, and the latter is one of the simplest simulations a user might make. Another scene consists of a fire being pushed by a directional force from the right and has a resolution of 256^3. Its purpose is to show how all the solvers are tested on a much larger grid size, making it a good way to compare the different solvers' ability to distribute residuals across a larger number of grid cells. A second feature of this scene is that of mass conservation. Since the main purpose of doing the pressure projection step is to enforce the incompressibility condition (mass conservation) we need to verify that the models are indeed capable of handling this as closely as possible to the multigrid solution. The final scene will be a tornado scene which serves as a stress test. The purpose of this scene is specifically to gauge how well the multiscale features of the pressure field are resolved by the different models.

Figure 8.2: Heatmap

The methodology for comparison will be as follows. First we compare the mean maximum absolute residual over 200 frames to get a metric for how good the different methods and models are at reducing the maximum residual across the simulation grid. Next for a structural analysis we generate three fully rendered images to showcase what the scene looks like when simulated using Jacobi, Gauss-Seidel and MultiGrid methods[2], with the same parameters (number of iterations/v-cycles) as determined in the performance tuning section (8.1). These images will serve as context when comparing the pressure fields generated by all the models and solvers. Generally, we will see that the multigrid produces the best visual quality (in terms of detail and fluid like features of the flow). Thus, we use the multigrid pressure field as the reference against which all other methods are compared. A heatmap (figure 8.2) is applied on the result of equation 8.1.

$$|Pressure - MultiGridPressure| \tag{8.1}$$

[2]In some scenes it may be hard to gauge the precise differences in quality but when viewed in motion the MultiGrid simulation produces the best visual quality

Random Scene

This scene consists of two cuboids and a sphere emitting our fuel, temperature and smoke. A planar force located towards the top of the grid pushes towards the bottom right at an angle. Amongst the machine learning models the MG small model has the lowest maximum residual by a small margin but all models with the exception of the FluidNet small model have comparable residual reduction in this scene. The pressure fields were scaled up by a factor of 1.5 to make the differences clearer.

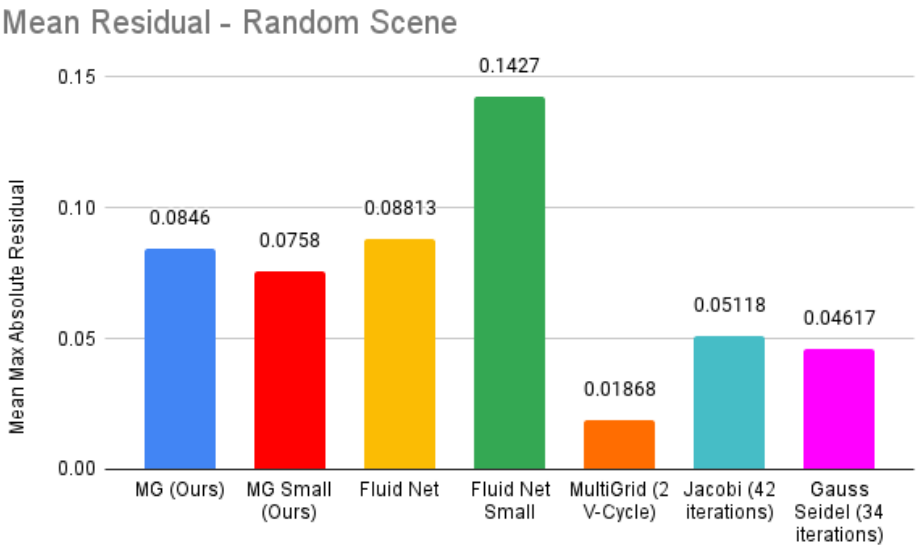
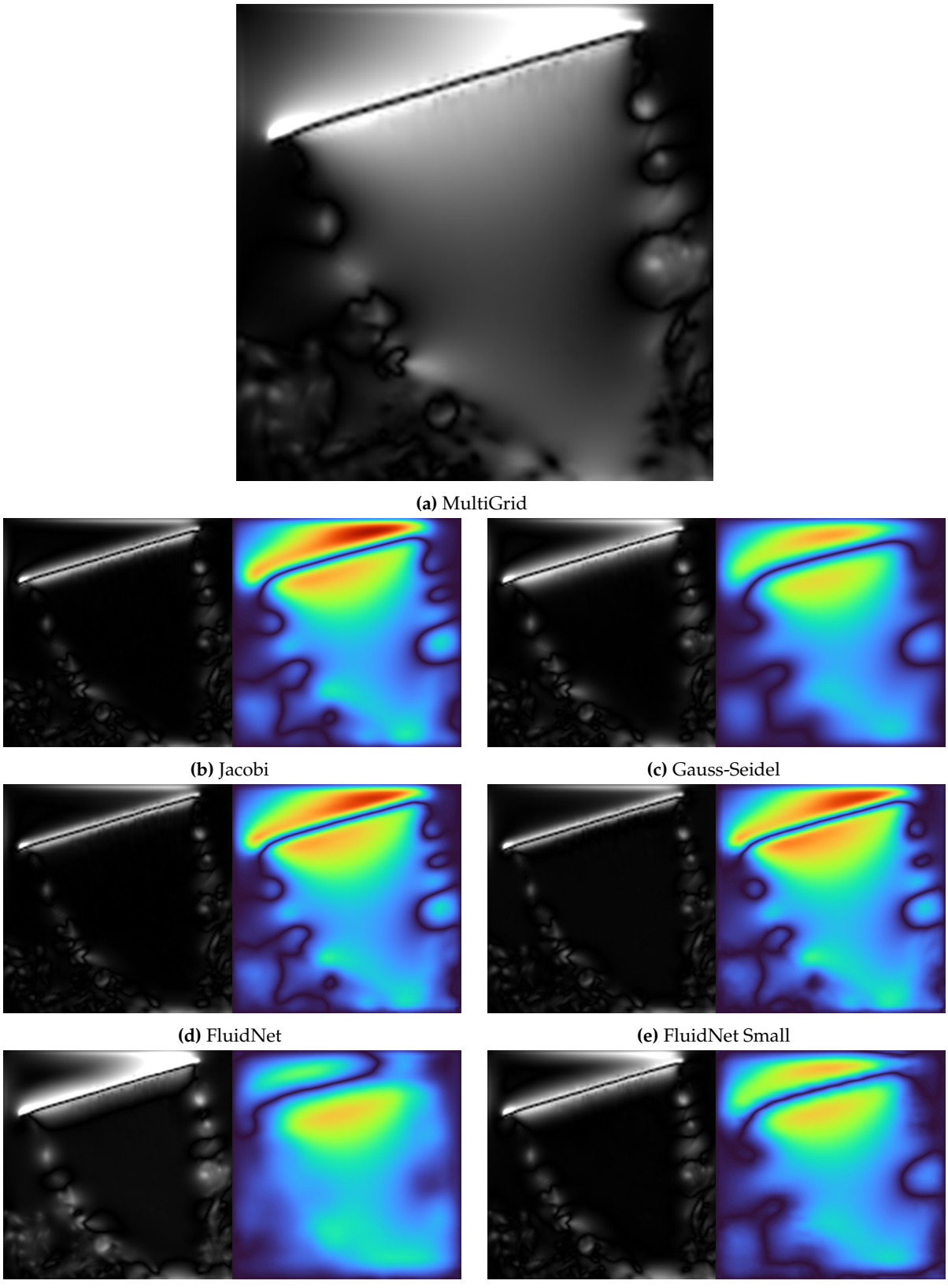


Figure 8.3: Random Scene: Mean Maximum Absolute Residual (200 frames)

(a) Jacobi **(b)** Gauss-Seidel **(c)** MultiGrid

Figure 8.4: Random Scene: Render Comparison



(a) MultiGrid

(b) Jacobi

(c) Gauss-Seidel

(d) FluidNet

(e) FluidNet Small

(f) MGML (Ours)

(g) MGML Small (Ours)

Figure 8.5: Random Scene: Pressure Field and diff against MultiGrid

Simple Fire

This scene consists of a hemisphere which emits some fuel, temperature and smoke which then combusts to form a simple fire. No additional forces are applied. Amongst the machine learning models the MG model shows the best overall residual reduction. Our small model produces roughly the same maximum residual as the FluidNet model. The pressure fields were scaled up by a factor of 6 to make the differences clearer.

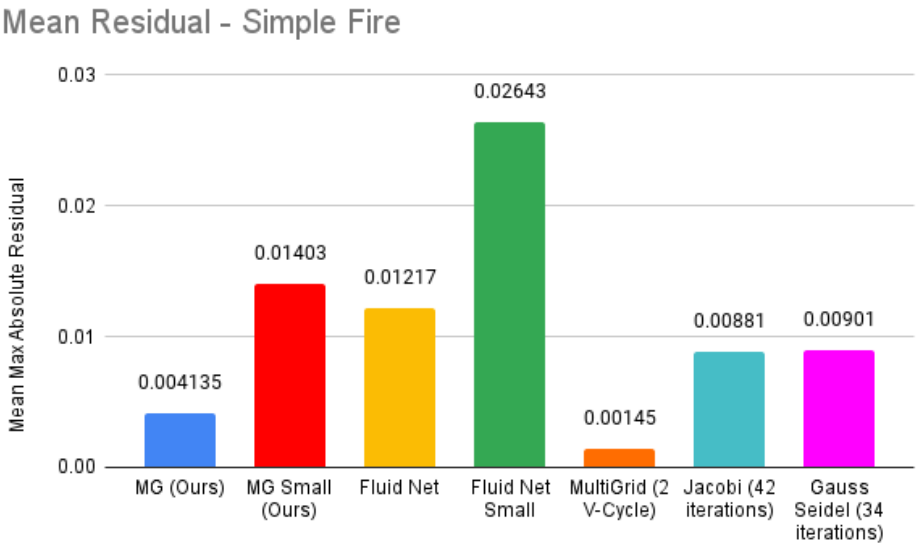
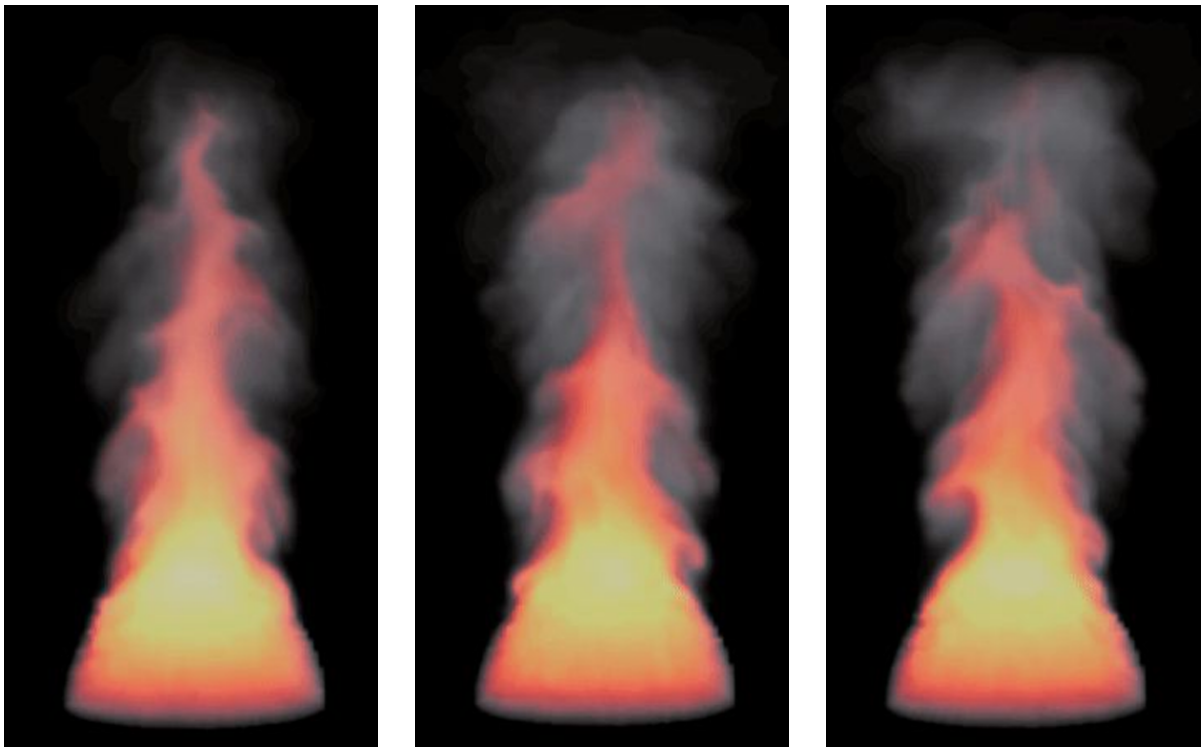
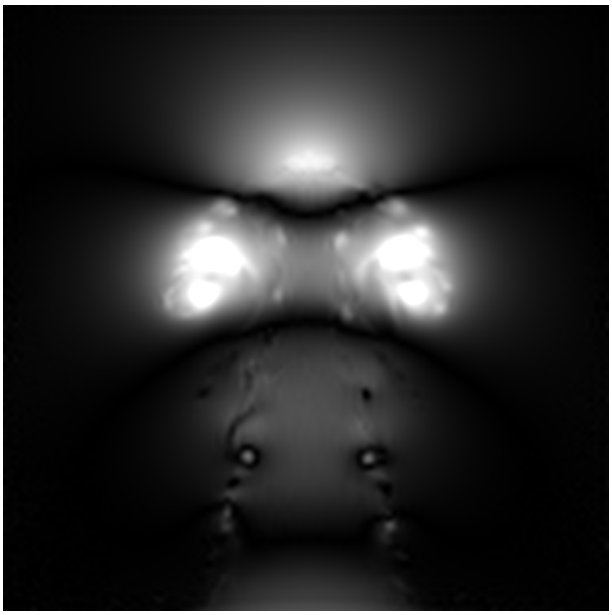


Figure 8.6: Simple Fire: Mean Maximum Absolute Residual (200 frames)

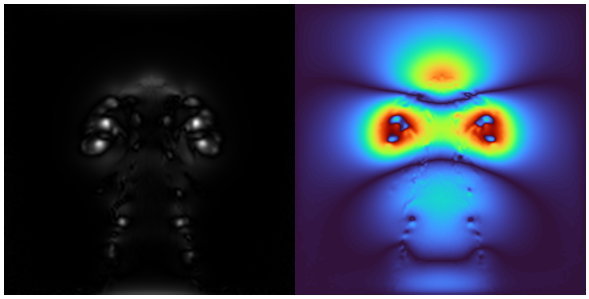


(a) Jacobi **(b)** Gauss-Seidel **(c)** MultiGrid

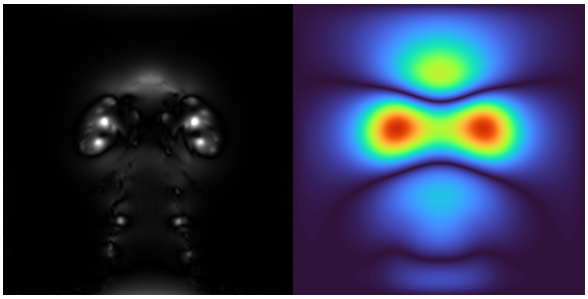
Figure 8.7: Simple Fire: Render Comparison



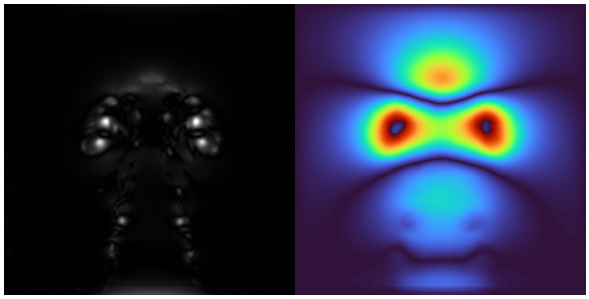
(a) MultiGrid



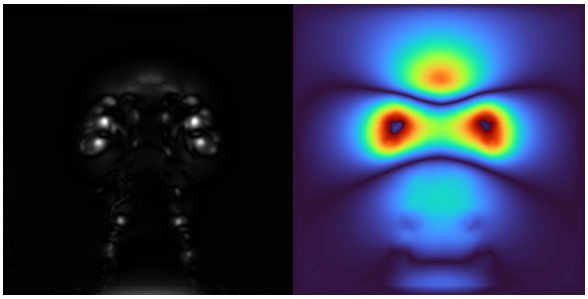
(b) Jacobi



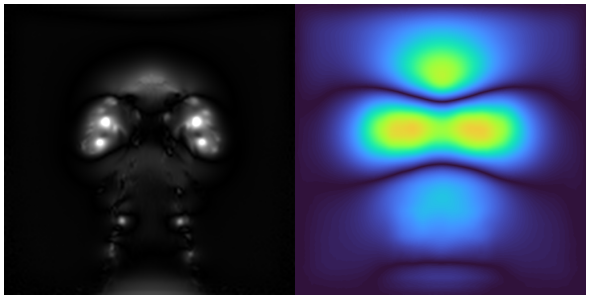
(c) Gauss-Seidel



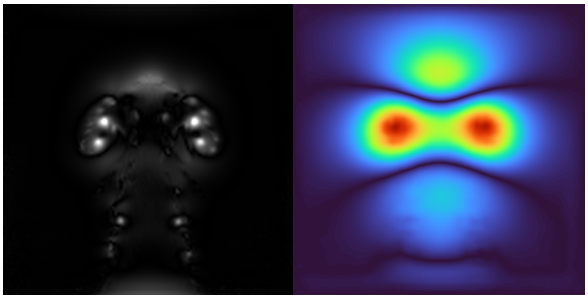
(d) FluidNet



(e) FluidNet Small



(f) MGML (Ours)



(g) MGML Small (Ours)

Figure 8.8: Simple Fire: Pressure Field and diff against MultiGrid

Fire and Wind

This scene consists of a cuboid which emits some fuel, temperature and smoke. A planar force is applied in the rightward direction causing the circular cutout in the rising fire. Amongst the machine learning models the MG model (ours) shows the best overall residual reduction almost to its small counterpart. Both variants of the FluidNet model have the highest residuals. The pressure fields were scaled by a factor of 6 to make the differences clearer.

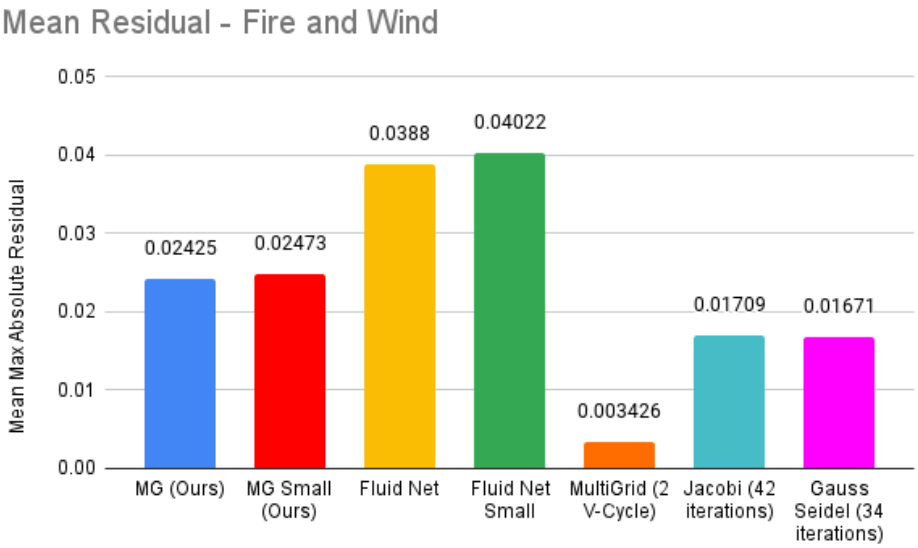
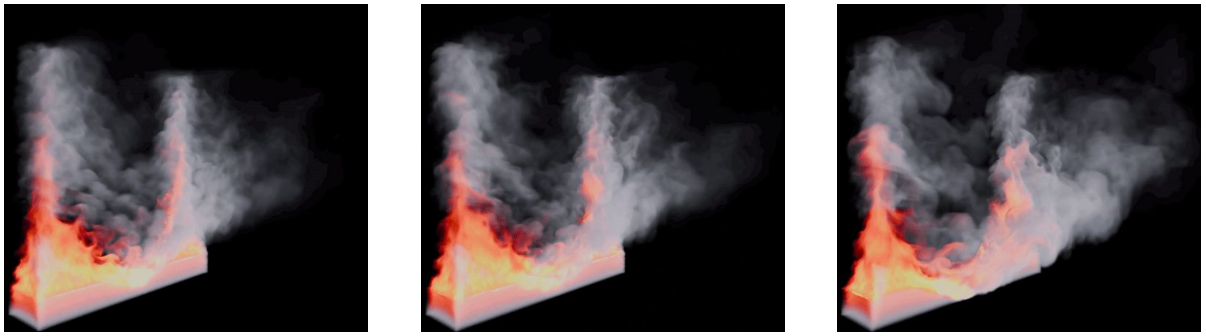
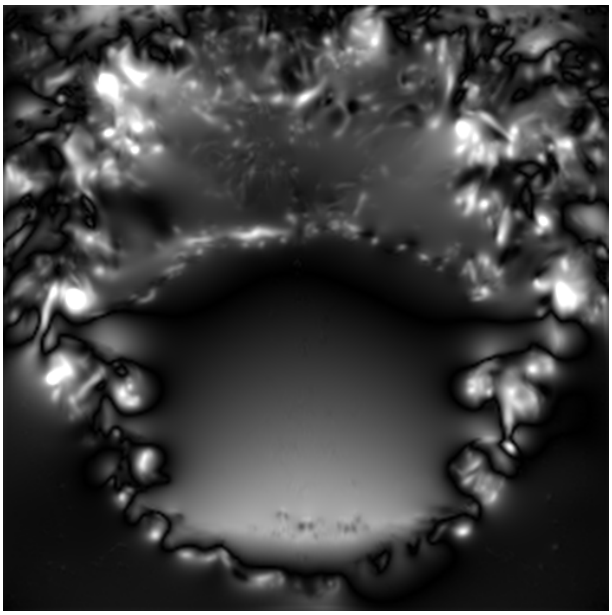


Figure 8.9: Fire and Wind: Mean Maximum Absolute Residual (200 frames)

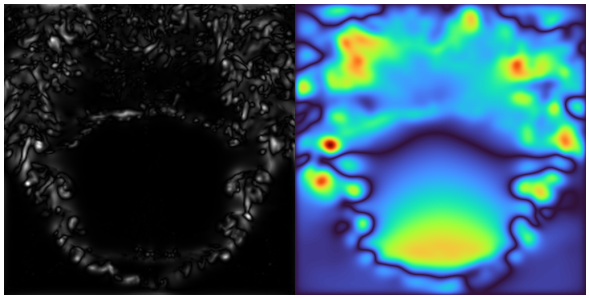


(a) Jacobi **(b)** Gauss-Seidel **(c)** MultiGrid

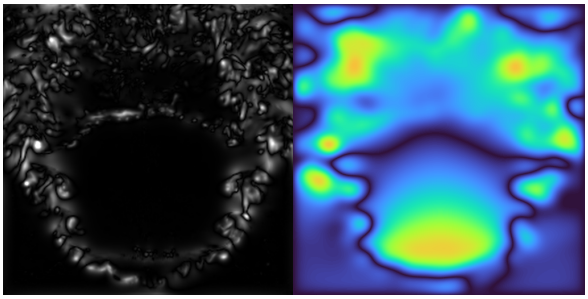
Figure 8.10: Fire and Wind: Render Comparison



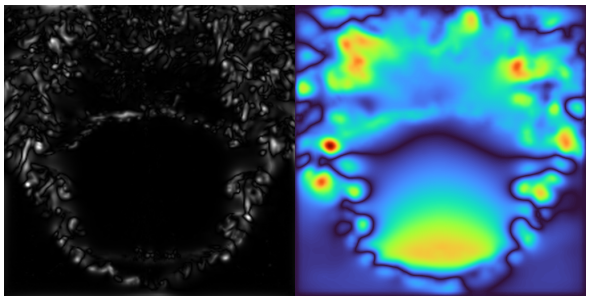
(a) MultiGrid



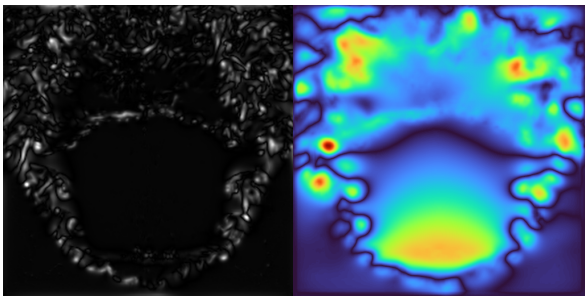
(b) Jacobi



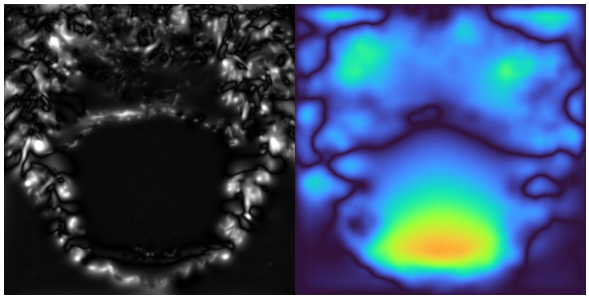
(c) Gauss-Seidel



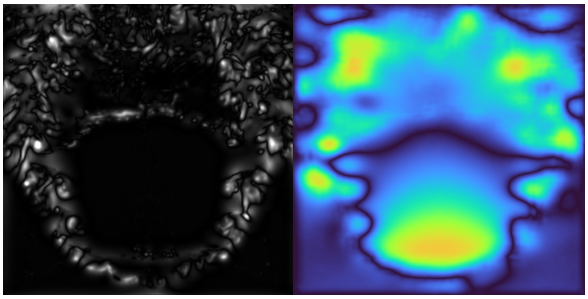
(d) FluidNet



(e) FluidNet Small



(f) MGML (Ours)



(g) MGML Small (Ours)

Figure 8.11: Fire and Wind (256^3): Pressure Field and diff against MultiGrid

Tornado

This scene consists of a cylinder which emits some fuel, temperature and smoke. A line force is applied in the upward direction along with a circular force around the line, resulting in a corkscrew shaped force. Amongst the machine learning models the MG model shows the best overall residual reduction. Our small model produces the highest maximum residual. The pressure fields were scaled by a factor of 0.5 to make the differences clearer.

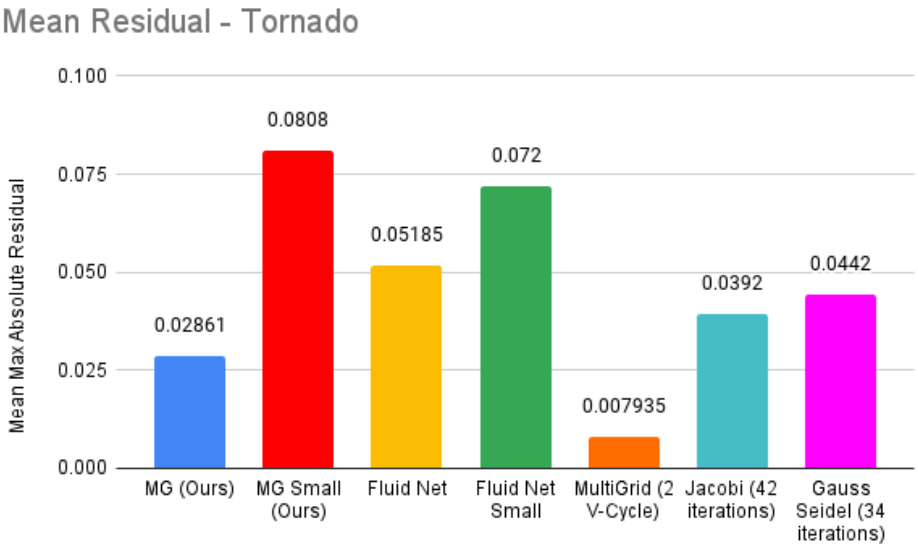
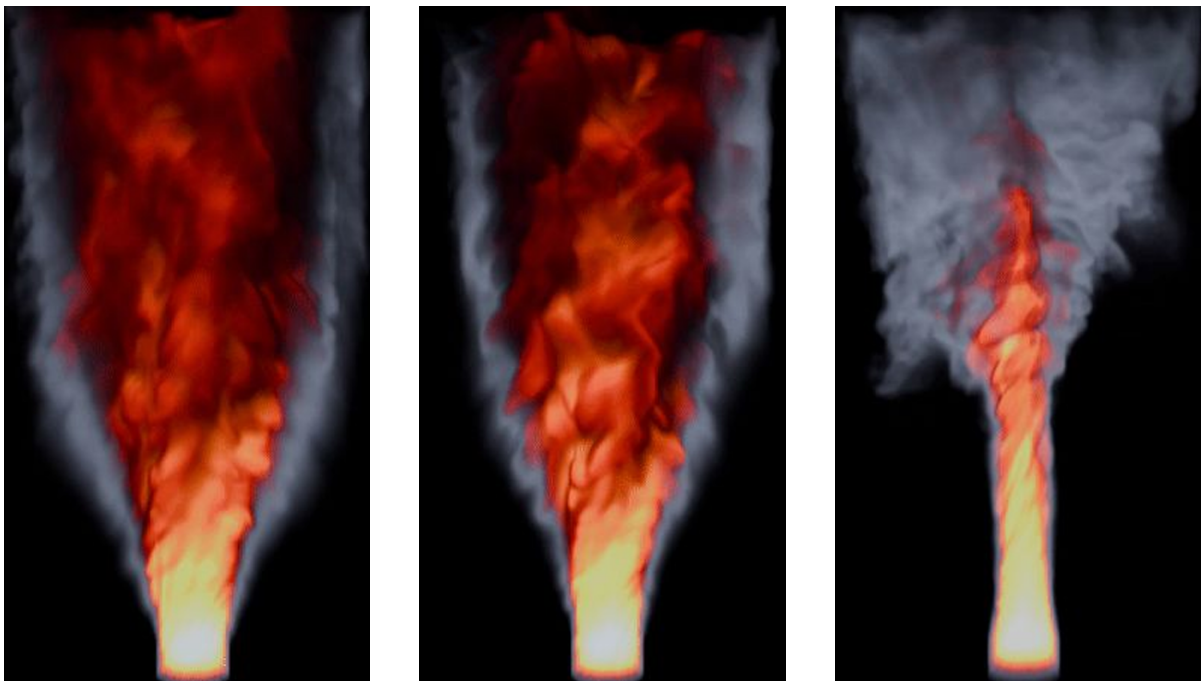
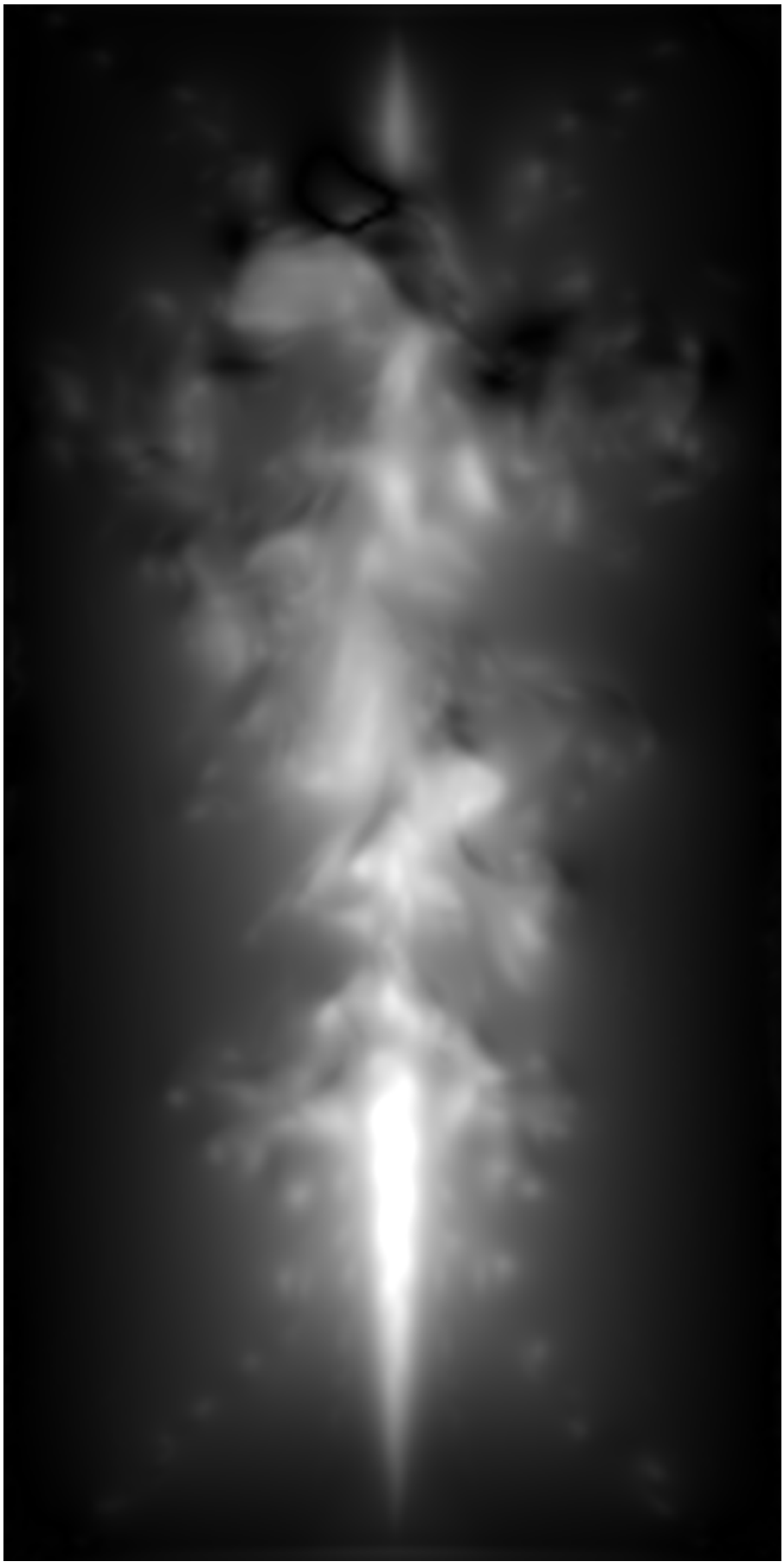


Figure 8.12: Tornado: Mean Maximum Absolute Residual (200 frames)

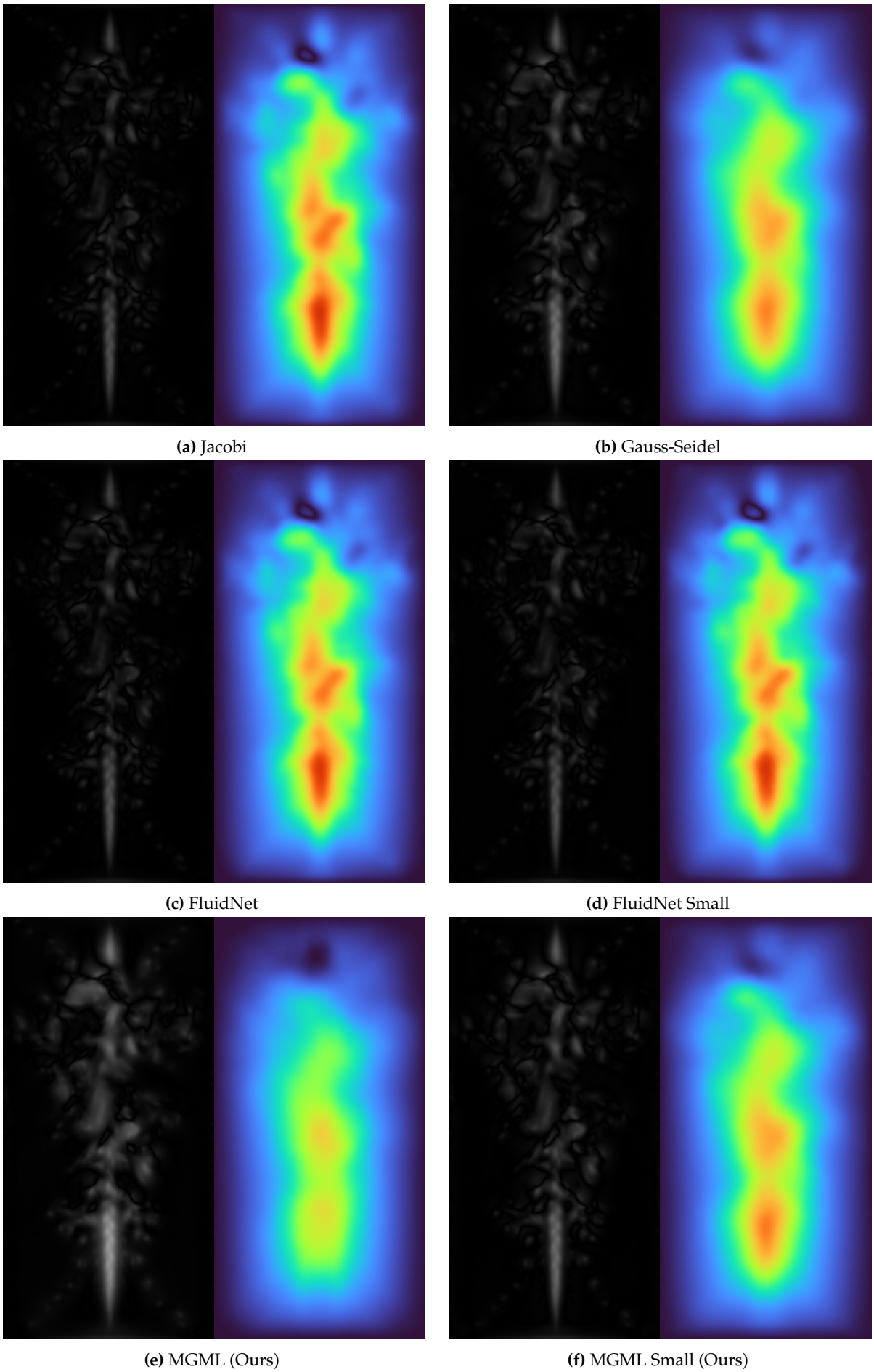


(a) Jacobi **(b)** Gauss Seidel **(c)** MultiGrid

Figure 8.13: Tornado: Render Comparison



(a) MultiGrid



(a) Jacobi

(b) Gauss-Seidel

(c) FluidNet

(d) FluidNet Small

(e) MGML (Ours)

(f) MGML Small (Ours)

Figure 8.15: Tornado: Pressure Field and diff against MultiGrid

In all four of the previous scenes we see that the multigrid solver at 2 V-cycles shows the greatest residual reduction by a significant margin across the board. In the static renders it's not always clear like in the simple fire scene (figure 8.7) which solver produces the best results, but as a general trend we do see that the solutions generated by the multigrid method have more fluid like features such as billowing effects and better rolling motions caused by the multiscale vortices that it manages to resolve. Examining the fire and wind scene (figure 8.10), we see that the smoke travels not only further but has more smoke volume at a greater distance from the source of the fire than both the Jacobi and Gauss-Seidel solutions, this shows that it is able to retain more of the smoke by correctly resolving the rolling motions caused by the wind and properly correcting for regions of negative divergence. The random scene (figure 8.4) is a little harder to reason about, but it does demonstrate a much more lively combustion with many wisps of fire being advected across larger distances than in the other two solutions. Finally, the tornado scene (figure 8.13) provides the starkest contrast between solvers where the Jacobi and Gauss-Seidel solvers demonstrate significant outward divergence from the source of the spiral force causing the volume of the smoke to increase significantly while simultaneously failing to properly capture the spiral motion.

Since the multigrid method provides both the best visual features in an active simulation and the lowest residuals consistently, the choice to compare all other methods against it, specifically for structural similarity seems, to us, fair. In all scenes the pressure field produced by the MultiGrid solver resolves not only the same small scale features as seen in the Jacobi and Gauss-Seidel solutions, but has around these features, medium and large scale halos which correspond directly to the multiscale features we see in the simulations. By capturing the differences in a heatmap it becomes easier to reason about which methods provide the closest structural similarities to the MultiGrid solution.

Looking first amongst the machine learning based methods, the fields produced by both our full sized MGML model and small models are much lower in magnitude than both FluidNet variants. The full model especially shows its capability of capturing at least the medium scale details with much more consistency than the other models. If we include the Jacobi and Gauss-Seidel solutions, the MGML small model very closely matches the Gauss-Seidel Solution in each scene and the full MGML model consistently produces smoother and lower magnitude diffs than all other methods. The FluidNet models fall short of the Gauss-Seidel solutions across the board only slightly beating the Jacobi solutions in some cases (the simple fire scene is the only place where there is a clear difference between the Jacobi and FluidNet outputs). The most impressive scene however remains the Tornado, where we see that our MGML model produces the closest match to the multigrid solution along with a lower residual than any other model or solver. This clearly demonstrates that especially in scenes where the multiscale features of the pressure field are essential, the multigrid inspired architecture beats by a significant margin the solutions provided by the other methods.

8.3 Conclusions

Having performed all the above experiments we can answer the questions posed in chapter 4.

Research Question: Accelerated Solution of the Pressure equation

We achieved significant improvements in execution time over the previous state of the art in machine learning methods used to solve the pressure Poisson equation. These improvements are even more significant in our reduced size model executing over 2.5 times faster at just 1.34 milliseconds per inference. Even our full size model executes at nearly the same speed as the reduced size FluidNet model while vastly outperforming it in terms of quality.

If we extend our comparison to the models' ability to reduce the maximum residual across the simulation domain, the models proposed in this thesis do demonstrate better behaviour across the entire validation set, and also in most scenes we tested manually. The full MGML model outperforms the Jacobi and Gauss-Seidel methods in some scenes by this metric.

A structural comparison shows that despite the models developed as part of this thesis are far more capable of capturing the multiscale features in the pressure fields than any of the other methods we compared against. Even the reduced size model is at its worst roughly equivalent to the Gauss-Seidel solutions whereas the FluidNet solution only matches the Jacobi solutions.

Despite these significant improvement, the multigrid method remains the clear choice in a practical real-time application where visual quality is of paramount importance. It's ability to reduce the residual quickly and resolve features across small and large scales gives it a significant edge on the other methods, and even further improvements would be needed to make machine learning based methods a better alternative in the general case.

Sub Research Question: Using Domain knowledge Effectively

By first implementing several solvers, and exploring the range of fluid simulation and various configurations and scenes, we were able to better grasp some underlying principles behind visually pleasing fluid simulations. By taking significant inspiration from the multigrid solver, we were able to carefully craft the model architecture to more effectively capture the multiscale features of the pressure field compared to the FluidNet architecture. A happy consequence of our restructuring of the architecutre was faster execution times and better residual and structural properties as discussed above (subsection 8.3).

A smaller improvement was made in the loss function where we directly use the residual formulation of the loss, reducing the amount of data required when training by a factor of 3. While not so important in a real world deployment where the model only needs to be trained once, in a limited time research context, the reduced data and increased simplicity proved quite beneficial.

9. Future Work

The methods and architectures in this thesis provide a solid foundation, on top of which a wide variety of ideas can be implemented. While we have achieved improvements in inference time and accuracy, there remains room for improvement in both these areas. In this chapter we provide, what we think are, in order of increasing difficulty, a series of ideas and extensions that can be used to improve the usability of machine learning methods for the pressure projection problem.

Increasing Model Depth Vertically

As presented in figure 7.3 the MGML model operates at 3 distinct resolutions, a full resolution grid, a half and finally a quarter resolution grid. If the input were of size, 128^4, the smallest grid would be 32^3. In the multigrid implementation, however we choose to go all the way down to an 8^3 grid at the lowest resolution. The lower the resolution, the larger the scale of feature we are able to resolve. By adding further verticality to the model architecture and downscaling up to 2 more times, would likely allow the model to significantly improve its structural capabilities, at nearly no performance cost, since the additional convolutions would be performed on very small grids in comparison to the base resolution grid.

Initial Estimate

The model takes as input, the divergence of our velocity field, post the addition of forces. If however we were to include the solution of the pressure field from a previous frame as an 'initial guess', we give the model more information to work with. The inspiration behind this idea, is the drastically improved convergence of the Jacobi and Gauss-Seidel methods when provided with the previous frame's pressure solution. Since the timesteps we operate on are so short, the change between pressure fields isn't too drastic. Therefore, a few iterations of the iterative solvers is able to correct for the imbalances post advection.

This extension would also potentially allow turning the learning based method into an iterative solver where the output could be fed through the network until a sufficient accuracy is reached. Of course, this remains limited by our computational budget.

More Flexible Activation Functions

Recent research has demonstrated that traditional activation functions, such as the ReLU, tanh, and sigmoid functions, can be improved upon by replacing them with a learnable activation. Kolmogorov Arnold Network[38] employ such a technique, where the activation functions are replaced by splines with learnable control points. The use of these ideas has the potential of greatly reducing network size. If successfully applied to the CNN's described in this paper, we can expect a reduction in the number of learnable parameters, while retaining or perhaps even improving network accuracy even further.

Including Advection in the model

The LSTM architecture has shown great success in time series predictions. If we can combine a pressure projection network, that effectively produces divergence free velocity fields, with an LSTM based advection, we may be able to further consolidate more of the fluid simulation into a single network. This would have the benefit of retaining interactivity by keeping the source and force addition steps fully controllable.

Upscaling

Neural Networks, specifically CNNs have been shown to excel at upscaling tasks for real-time applications. Some such examples include Nvidia's DLSS (Deep Learning Super Sampling), and AMD's Fidelity FX Super Resolution, that are widely included in most AAA games. Research in fluid simulation has also shown success in using learning based upscalers[34] to increase the detail in a simulation. If these models can be made to run in real-time, a lower resolution simulation could be simply upscaled at the end of each simulation step, potentially reducing the computational cost.

Bibliography

[1] J. Stam, "Stable fluids," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, ser. SIGGRAPH '99, ACM Press, 1999. DOI: 10.1145/311535.311548. [Online]. Available: http://dx.doi.org/10.1145/311535.311548.

[2] B. E. Feldman, J. F. O'Brien, and O. Arikan, "Animating suspended particle explosions," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 708–715, Jul. 2003, ISSN: 1557-7368. DOI: 10.1145/882262.882336. [Online]. Available: http://dx.doi.org/10.1145/882262.882336.

[3] R. Miyazaki, Y. Dobashi, and T. Nishita, "Simulation of cumuliform clouds based on computational fluid dynamics," Jan. 2002.

[4] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH01, ACM, Aug. 2001. DOI: 10.1145/383259.383260. [Online]. Available: http://dx.doi.org/10.1145/383259.383260.

[5] J. Stam, "Real-time fluid dynamics for games," May 2003.

[6] M. Bonner, "Compressible subsonic flow on a staggered grid," Apr. 2011.

[7] R. Bridson, *Fluid Simulation*. USA: A. K. Peters, Ltd., 2008, ISBN: 1568813260.

[8] *Embergen*, https://jangafx.com/software/embergen, [Accessed 25-07-2024].

[9] M. Vassvik, *Realtime fluid simulation: Projection*, Apr. 2022. [Online]. Available: https://gist.github.com/vassvik/f06a453c18eae03a9ad4dc8cc011d2dc.

[10] K. Museth, "Vdb: High-resolution sparse volumes with dynamic topology," *ACM Transactions on Graphics*, vol. 32, no. 3, pp. 1–22, Jun. 2013, ISSN: 1557-7368. DOI: 10.1145/2487228.2487235. [Online]. Available: http://dx.doi.org/10.1145/2487228.2487235.

[11] F. H. Harlow and J. E. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *The Physics of Fluids*, vol. 8, no. 12, pp. 2182–2189, Dec. 1965, ISSN: 0031-9171. DOI: 10.1063/1.1761178. [Online]. Available: http://dx.doi.org/10.1063/1.1761178.

[12] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis, "Spgrid: A sparse paged grid structure applied to adaptive smoke simulation," *ACM Transactions on Graphics*, vol. 33, no. 6, pp. 1–12, Nov. 2014, ISSN: 1557-7368. DOI: 10.1145/2661229.2661269. [Online]. Available: http://dx.doi.org/10.1145/2661229.2661269.

[13] W. Raateland, T. Hädrich, J. A. A. Herrera, *et al.*, "Dcgrid: An adaptive grid structure for memory-constrained fluid simulation on the gpu," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 5, no. 1, pp. 1–14, May 2022, ISSN: 2577-6193. DOI: 10.1145/3522608. [Online]. Available: http://dx.doi.org/10.1145/3522608.

[14] T. L. van Wingerden, "Real-time ray tracing and editing of large voxel scenes," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:53850534.

[15] B. Kim, Y. Liu, I. Llamas, and J. Rossignac, "FlowFixer: Using BFECC for Fluid Simulation," in *Eurographics Workshop on Natural Phenomena*, P. Poulin and E. Galin, Eds., The Eurographics Association, 2005, ISBN: 3-905673-29-0. DOI: 10.2312/NPH/NPH05/051-056.

[16] S. Brunton, *Runge kutta integrators*, https://www.youtube.com/watch?v=HOWJp8NV5xU, [Accessed 25-07-2024].

[17] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, "An unconditionally stable maccormack method," *Journal of Scientific Computing*, vol. 35, no. 2–3, pp. 350–371, Nov.

2007, ISSN: 1573-7691. DOI: 10.1007/s10915-007-9166-4. [Online]. Available: http://dx.doi.org/10.1007/s10915-007-9166-4.

[18] X. Zhang, R. Bridson, and C. Greif, "Restoring the missing vorticity in advection-projection fluid solvers," *ACM Transactions on Graphics*, vol. 34, no. 4, Jul. 2015, ISSN: 1557-7368. DOI: 10.1145/2766982. [Online]. Available: http://dx.doi.org/10.1145/2766982.

[19] J. Zehnder, R. Narain, and B. Thomaszewski, "An advection-reflection solver for detail-preserving fluid simulation," *ACM Transactions on Graphics*, vol. 37, no. 4, pp. 1–8, Jul. 2018, ISSN: 1557-7368. DOI: 10.1145/3197517.3201324. [Online]. Available: http://dx.doi.org/10.1145/3197517.3201324.

[20] G. Amador and A. Gomes, "Linear solvers for stable fluids: Gpu vs cpu," in *ACTAS DO 17º ENCONTRO PORTUGUÊS DE COMPUTAÇÃO GRÁFICA*, A. Coelho and A. P. Cláudio, Eds., The Eurographics Association, 2021, ISBN: 978-3-03868-154-0. DOI: 10.2312/pt.20091234.

[21] C. Dick, M. Rogowsky, and R. Westermann, "Solving the fluid pressure poisson equation using multigrid—evaluation and improvements," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 11, pp. 2480–2492, Nov. 2016, ISSN: 1077-2626. DOI: 10.1109/tvcg.2015.2511734. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2015.2511734.

[22] W. Engel, *GPU Pro 360 Guide to Rendering*, 1st. USA: A. K. Peters, Ltd., 2018, ISBN: 0815365519.

[23] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '10, Madrid, Spain: Eurographics Association, 2010, pp. 65–74.

[24] N. Sakharnykh, *High-performance geometric multi-grid with gpu acceleration*, Feb. 2016. [Online]. Available: https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/.

[25] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," USA, Tech. Rep., 1994.

[26] N. Foster and R. Fedkiw, "Practical animation of liquids," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH01, ACM, Aug. 2001. DOI: 10.1145/383259.383261. [Online]. Available: http://dx.doi.org/10.1145/383259.383261.

[27] N. Chentanez and M. Mueller-Fischer, "A multigrid fluid pressure solver handling separating solid boundary conditions," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 8, pp. 1191–1201, Aug. 2012, ISSN: 1077-2626. DOI: 10.1109/tvcg.2012.86. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2012.86.

[28] J. Jiang, X. Shen, Y. Gong, *et al.*, "A second-order explicit pressure projection method for eulerian fluid simulation," *Computer Graphics Forum*, vol. 41, no. 8, pp. 95–105, Dec. 2022, ISSN: 1467-8659. DOI: 10.1111/cgf.14627. [Online]. Available: http://dx.doi.org/10.1111/cgf.14627.

[29] R. Bridson, J. Houriham, and M. Nordenstam, "Curl-noise for procedural fluid flow," *ACM Transactions on Graphics*, vol. 26, no. 3, p. 46, Jul. 2007, ISSN: 1557-7368. DOI: 10.1145/1276377.1276435. [Online]. Available: http://dx.doi.org/10.1145/1276377.1276435.

[30] S. Rødal, G. Storli, and O. E. Gundersen, "Physically Based Simulation and Visualization of Fire in Real-Time using the GPU," in *Theory and Practice of Computer Graphics 2006*, L. M. Lever and M. McDerby, Eds., The Eurographics Association, 2006, ISBN: 3-905673-59-2. DOI: 10.2312/LocalChapterEvents/TPCG/TPCG06/013-020.

[31] EpicGames, *Niagara fluids*, https://dev.epicgames.com/community/learning/paths/mZ/unreal-engine-niagara-fluids, [Accessed 25-07-2024].

[32] EpicGames, *Unrealengine*, https://www.unrealengine.com/en-US, [Accessed 25-07-2024].

[33] R. Wang and R. Yu, *Physics-guided deep learning for dynamical systems: A survey*, 2021. DOI: 10.48550/ARXIV.2107.01272. [Online]. Available: https://arxiv.org/abs/2107.01272.

[34] C. Li, S. Qiu, C. Wang, and H. Qin, "Learning physical parameters and detail enhancement for gaseous scene design based on data guidance," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 10, pp. 3867–3880, Oct. 2021, ISSN: 2160-9306. DOI: 10.1109/tvcg.2020.2991217. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2020.2991217.

[35] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating eulerian fluid simulation with convolutional networks," Jul. 2017.

[36] D. J. Tritton, *Physical Fluid Dynamics*, 2nd ed. Oxford, England: Clarendon Press, Sep. 1988.

[37] G. D. Weymouth, "Data-driven multi-grid solver for accelerated pressure projection," *Computers and Fluids*, vol. 246, p. 105 620, Oct. 2022, ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2022.105620. [Online]. Available: http://dx.doi.org/10.1016/j.compfluid.2022.105620.

[38] Z. Liu, Y. Wang, S. Vaidya, *et al.*, *Kan: Kolmogorov-arnold networks*, 2024. DOI: 10.48550/ARXIV.2404.19756. [Online]. Available: https://arxiv.org/abs/2404.19756.