Smoke Simulation for Fire Engineering using CUDA

Masters thesis

by

Stefan Glimberg

glimberg@diku.dk

Department of Computer Science
University of Copenhagen
June 1, 2009
Abstract

Computational solutions to the Navier-Stokes equations for fluid dynamics has become an increasingly useful tool for engineering purposes. However, solving fluid dynamics accurately requires a considerable amount of time and computer resources.

In this thesis we investigate the possibilities of interactive smoke simulation for engineering purposes, using the recently released programming model CUDA. CUDA delivers a programming API that utilize the highly parallel architecture of GPUs. Our computational model is based on a fractional step method to solve the partial differential fractions of the Navier-Stokes equations. A multigrid method has been implemented to solve the pressure Poisson equation. We compare the multigrid solver with a classical Jacobi solver and show that it is both more efficient, accurate and creates better visual details.

Resumé

Beregningsmodeller til løsning af Navier-Stokes ligningerne er i stigende grad blevet et brugbart værktøj indenfor ingeniør-relateret arbejde. Disse modeller kræver dog et massivt forbrug af computerressourcer samt et højt tidsforbrug.

I dette speciale undersøger vi mulighederne for interaktiv røgsimulering til brandtekniske simuleringer, ved brug at den nyudgivne programmeringsmodel CUDA. CUDA stiller et API til rådighed der udnytter den parallele arkitektur på grafikkortet. Vores beregningsmodel baserer sig på en fractional step metode til løsning af de partielle differential led fra Navier-Stokes ligningerne. Løsningen af tryklinningssystem er blevet implementeret som en multigrid metode. Vi sammenligner multigrid metoden med en klassisk Jacobi løser og viser at den er både mere effektiv, nøjagtig og giver bedre visuelle detaljer.

# Contents

1 Interactive Smoke Simulation for Risk Analysis .......................... 5  
1.1 Rambøll Collaboration ................................................. 6  
1.2 Related Work .......................................................... 7  
1.3 Smoke Simulation using CUDA ...................................... 7  

2 Mathematical Introduction ............................................... 9  
2.1 Scalar, Vector and Field Notation ..................................... 9  
2.2 The Nabla Operator ..................................................... 9  

3 Fluid Dynamics ............................................................... 13  
3.1 Fluid Properties .......................................................... 13  
3.2 Eulerian vs Lagrangian Methods ....................................... 15  
3.3 Fluid Types .............................................................. 16  
3.4 The Navier-Stokes Equations .......................................... 17  
3.5 Conservation Laws ..................................................... 18  

4 Parallel Computing using CUDA .......................................... 23  
4.1 Programming Model ..................................................... 23  
4.2 Numerical Precision .................................................... 30  

5 Discretization ................................................................. 31  
5.1 Finite Difference ....................................................... 33  

6 The Fractional Step Method ................................................ 35  
6.1 Navier-Stokes in Fractions ............................................. 35  
6.2 Solving Linear Systems ................................................ 40  
6.3 Scalar Fields ............................................................ 48  
6.4 Boundary Conditions .................................................. 49  

7 The Application ............................................................... 55  
7.1 Setting up Grids and Blocks ........................................... 55  
7.2 FDS Input File Format .................................................. 57  
7.3 The CFD Solver .......................................................... 58  
7.4 Visualization ............................................................. 58
Chapter 0

CONTENTS

7.5 Data Plotting .......................................................... 60

8 Results ................................................................. 61
  8.1 Fractional Tests ....................................................... 61
  8.2 Case Studies .......................................................... 66
  8.3 Summarizing the Results .......................................... 81

9 Conclusion .............................................................. 85
  9.1 Future Work .......................................................... 86

A Rambøll Evaluation ............................................... 91
Chapter 1

Interactive Smoke Simulation for Risk Analysis

In this thesis we present a Computational Fluid Dynamics (CFD) solver for smoke propagation, using the parallel architecture of graphics hardware. The purpose of the thesis has been to present a solution based on fast approximation techniques and evaluate the usability from an engineering point of view. CFD tools help engineers to analyse security and risks at fire accidents. Especially the evolution of smoke thickness and temperatures at a fire scene is important knowledge for building engineers. Based on several case studies, discussed in the result section, our solver has shown to be an efficient tool for interactive smoke simulation. Even at high resolutions the solver performs at interactive rates on a single Graphical Processing Unit (GPU). The solver was implemented using the fairly new programming model CUDA (former known as Compute Unified Device Architecture). CUDA has become an attractive and fast growing programming tool within the last year. CUDA is an extended C compiler that compiles code that can be executed on GPUs. The parallel architecture of the GPU makes it an excellent computational unit for tasks like CFD. In Chapter 4 we give a presentation of the CUDA programming model and Chapter 7 describes the complete CFD solver application. A fractional step method was utilized and implemented to solve the Navier-Stokes equations on the GPU. Spatial and temporal differentiations are approximated using first order accurate finite differences. To improve accuracy and efficiency, we have implemented a multigrid solver for the linear pressure-Poisson equation. A detailed description of the solution model is presented in Chapter 6.
1.1 Rambøll Collaboration

The request to examine the computational possibilities of a GPU based CFD solver originates from Rambøll Denmark, an international corporation of consulting engineers. Currently the department of risk and safety analysis at Rambøll, uses both commercial and noncommercial CFD programs to simulate and evaluate fire risks in buildings. One downside of these programs is, that computations are so expensive, that simulations are not interactive. Instead off-line computations store simulation data in files for subsequent visualization. Large configurations require intense calculations and lots of hard disk space. The goal for Rambøll is to get an alternative tool that runs fast on low budget hardware configurations. Modern graphics hardware, delivers just these possibilities in connection with CUDA. Such fast approximation methods usually suffer from inaccuracies, nevertheless they might serve as a good pre-process simulation to the otherwise time consuming CFD programs.

During the time of the thesis, we have held meetings with Rambøll, to present and evaluate the results. These meetings have been used to decide what parts of the solver to be improved. Halfway through the thesis, we observed that the numerical method caused intense dissipation and loss of details. Convergence plots for the iterative Jacobi solver underlined that it did not perform very well. It was therefore chosen to be improved with a multigrid method. In Section 6.2.2 we present the multigrid technique and in Section 8.2.1 we compare the efficiency and quality of the Jacobi and the multigrid methods.

1.1.1 Fire Dynamics Simulator

One of the intensively used CFD simulators at Rambøll, is the noncommercial Fire Dynamics Simulator (FDS)\[24\]. FDS is an approved CFD solver for fire engineering, which we will use to compare our results to. FDS solves a Large-Eddie Simulation (LES) form of the Navier-Stokes equation. A rather advanced and complex approach, known to produce good results. In order to make our CFD solution as user-friendly as possible to Rambøll, we adopted a subset of the file input format used by FDS and extended it with two additional parameters. In Chapter 7.2 we present the FDS file format and describe the parameters that are supported by our CFD solver. Inspired by FDS, our solution are also capable of storing various fluid information into text files. In Section 8.2.5 we compare our results to similar FDS simulations.
1.2 Related Work

Within the last few years, several papers have been presented for solving CFD related problems using graphics hardware. [34] and [26] both present real-time simulators using shader techniques, with game environments as the target platform. Common for most of such papers are that they prefer visual quality on behalf of correct physical behavior of the flow. Therefore they use less accurate numerical models to speed up the calculations. [15] presents a GPU implementation of a two dimensional solver, based on the unconditional stable advection scheme, first presented by Stam in [36]. The solver was later extended to three dimensions by Harris et al. in [15]. It was also extended to handle dynamic boundaries. The foundation of our solver will be based party on these papers.

After NVIDIA presented CUDA, there has been a frequent update of publications on CFD related issues solved on GPUs. However, the most accurate and advanced CFD solvers are still based on off-line simulation. In [3], Bennetsen presents a survey of several modern CFD techniques and compares them to his own work in a set of test cases. A wide variety of test cases exist for CFD verification. In this thesis we have used the reference results presented by Ghia et al. in [38] and from Bennetsen in [3].

A vast amount of literature on fluid dynamics exists, we recommend [9] which gives a broad introduction to computational methods for solving the Navier-Stokes equations. For a more general introduction to fluids we recommend [21].

The authors background takes off in the fields of computer graphics and physics-based animation. In previous projects we have been working with game engines [12][13] and post-processing effects using shader programming [14]. Working with shaders has served as a good background knowledge to the GPU architecture and parallel computing. CUDA kernels have several similarities to pixel-shaders. For various topics on GPU based paradigms we recommend the book series [8][32][27].

1.3 Smoke Simulation using CUDA

After introducing the fractional step method and the application using CUDA, the CFD solver is evaluated in a set of test cases. Chapter 8 first evaluates the fractional parts of the solver and then we compare it to some more advanced test cases with reference material. The tests confirm that the solver works as expected and that it compares well to the reference results, especially when grid resolutions are kept high. Fortunately, the solver has no problem with simulating high resolutions at interactive speeds. For selected simulations, our solver performs more than 300 times faster compared to FDS simulations. Comparison between two test cases using the CUDA solver and FDS is also presented in the result section.

Improving the linear system solver with a multigrid technique has shown to enhance the details significantly. Furthermore, the multigrid solver is shown to im-
prove convergence and thus minimize divergence, compared to a simple Jacobi solver. Figure 1.1 gives two illustrative examples of smoke simulations using the CUDA solver.

Comments from Rambøll have been very optimistic and they expect to get great value of the CFD solver presented here. As part of the evaluation Jens Bennetsen wrote:

"The present CUDA solver show an initial implementation, which can be extended to included more advanced physical models. But at the current state it is fully useful for our business."

Figure 1.1: Left: Smoke raising around a sphere. Right: Smoke crippling up under stair steps. Both simulations are performed on 256x256x256 grid resolutions. At this resolution the solver calculates a frame every two seconds on a Tesla C1060 GPU.

On the enclosed DVD you will find demonstrative videos, FDS examples and more. We refer to the readme file on the DVD for detailed description of the content. The source code has not been attached due to requests from Rambøll. If you want access to the code, please write to the author for permission.
Chapter 2

Mathematical Introduction

Throughout this thesis, we present an intense amount of mathematical equations and notations. To ease the understanding and eliminate any possible misconceptions, this section summarizes the notation that will be used. We attempt to use a consistent and traditional notation, that should be familiar to students and scientists.

2.1 Scalar, Vector and Field Notation

When nothing else is mentioned, vectors will be real and three-dimensional. Bold face and small caps are used to denote vectors and small caps are used to denote scalars, i.e. \( \mathbf{u} = (u^x, u^y, u^z)^T \), \( \mathbf{u} \in \mathbb{R}^3 \), \( u^x, u^y, u^z \in \mathbb{R} \), where superscripts denote the vector components. Matrices will be in both bold and capital. A field is a mapping that associates values to a spatial point in space. In this thesis, all fields maps from three dimensional vectors \( \mathbf{x} = (x^x, x^y, x^z) \), to either scalar or vector values. If the mapping returns a scalar value \( (\mathbb{R}^3 \rightarrow \mathbb{R}) \), the field is called a scalar field. Likewise, a field that returns vectors \( (\mathbb{R}^3 \rightarrow \mathbb{R}^3) \), is called a vector field. The velocity vector at position \( \mathbf{x} = (x^x, x^y, x^z) \) at time \( t \) will be written as \( \mathbf{u}(\mathbf{x}, t) \). However, to reduce notation, we omit the parenthesis and the arguments when appropriate. The velocity field is therefore just written \( \mathbf{u} \).

Table 2.1 and 2.2 list the nomenclature used in this thesis.

2.2 The Nabla Operator

The nabla operator, \( \nabla \), is a differential operator appearing in the equations for fluid dynamics. It is sometimes written with a different notation in the literature, but the meaning is the same. The approximation of \( \nabla \) depends on the coordinate system.
# 2.2 The Nabla Operator

## Table 2.1: CFD related nomenclature. These are CFD related symbols that will be used throughout the thesis. \( \mathbf{F} \) is an exception to that a vector is lower case.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>The current time</td>
<td>( s )</td>
</tr>
<tr>
<td>( \Delta t )</td>
<td>The time step</td>
<td>( s )</td>
</tr>
<tr>
<td>( m )</td>
<td>Mass</td>
<td>( kg )</td>
</tr>
<tr>
<td>( \rho )</td>
<td>Density</td>
<td>( kg/m^3 )</td>
</tr>
<tr>
<td>( \mu )</td>
<td>Dynamic viscosity</td>
<td>( Pas )</td>
</tr>
<tr>
<td>( \nu )</td>
<td>Kinematic viscosity</td>
<td>( m^2/s )</td>
</tr>
<tr>
<td>( \mathbf{x} )</td>
<td>A spatial vector</td>
<td>( (m, m, m) )</td>
</tr>
<tr>
<td>( \mathbf{u} )</td>
<td>The velocity field</td>
<td>( (m/s, m/s, m/s) )</td>
</tr>
<tr>
<td>( p )</td>
<td>The pressure field</td>
<td>( N/m^2 )</td>
</tr>
<tr>
<td>( T )</td>
<td>The temperature field</td>
<td>( ^oC )</td>
</tr>
<tr>
<td>( \mathbf{F} )</td>
<td>The external force field</td>
<td>( (N, N, N) )</td>
</tr>
<tr>
<td>( \mathbf{f} )</td>
<td>The external force field per density</td>
<td>( (m^4/s^2, m^4/s^2, m^4/s^2) )</td>
</tr>
<tr>
<td>( g )</td>
<td>Gravitational constant</td>
<td>( (m/s^2, m/s^2, m/s^2) )</td>
</tr>
</tbody>
</table>

## Table 2.2: Grid related nomenclature. These are CFD related symbols that will be used throughout the thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I )</td>
<td>The grid width</td>
</tr>
<tr>
<td>( J )</td>
<td>The grid height</td>
</tr>
<tr>
<td>( K )</td>
<td>The grid depth</td>
</tr>
<tr>
<td>( N )</td>
<td>Total number of cells (I<em>J</em>K)</td>
</tr>
<tr>
<td>( \mathcal{G}_{I,J,K} )</td>
<td>A grid with dimension ( I, J, K )</td>
</tr>
<tr>
<td>( i )</td>
<td>Grid index in the x-direction</td>
</tr>
<tr>
<td>( j )</td>
<td>Grid index in the y-direction</td>
</tr>
<tr>
<td>( k )</td>
<td>Grid index in the z-direction</td>
</tr>
<tr>
<td>( \Delta x )</td>
<td>Cell size in x-direction</td>
</tr>
<tr>
<td>( \Delta y )</td>
<td>Cell size in y-direction</td>
</tr>
<tr>
<td>( \Delta z )</td>
<td>Cell size in z-direction</td>
</tr>
<tr>
<td>( \mathbf{u}_{i,j,k} )</td>
<td>The velocity at grid point ((i, j, k))</td>
</tr>
</tbody>
</table>

10
used. In this thesis we only use Cartesian coordinate systems, and only definitions herein are presented. For other system definitions, see [9]. The derivation of the Navier-Stokes equations using $\nabla$, are presented in Section 3.4. The challenge is how to transform $\nabla$ into a useful approximation in the discretized Cartesian coordinate system using finite difference methods. Several finite difference methods exist, some are presented in [31, 9]. In Section 5.1 we present the finite difference approximation used in this thesis and discuss the consequences of using such approximations.

$\nabla$ defines the vector of spatial derivatives:

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)^T \quad (2.1)$$

The following sections describes $\nabla$ as it appears in the Navier-Stokes equations. Transforming the nabla operator into a useful discrete representation using finite difference, is presented in Section 5.

2.2.1 The Gradient Operator

The gradient is defined from a scalar field, like $\nabla p$, and the result is a vector field. The gradient always points in the direction of the greatest rate of increase and the length represents the size of the increase rate. The definition of the gradient to a scalar field $p$ writes:

$$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}\right)^T \quad (2.2)$$

2.2.2 The Divergence Operator

Divergence is applied to a vector field, and measures the net flux through each point in the field, like $\nabla \cdot \mathbf{u}$. The divergence is a scalar field. An important property in CFD is incompressibility, which is often considered true. When the fluid is incompressible, the amount of fluid entering and leaving in each point must be equal. Hence, the divergence of the velocity field should be zero. This is further described in Section 3.3.1. The divergence operator applied to the velocity field $\mathbf{u}$ is defined as:

$$\nabla \cdot \mathbf{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} \quad (2.3)$$

2.2.3 The Laplacian Operator

The Laplacian operator is a second order differential operator defined over either a scalar or vector field. It is defined as the divergence of the gradient, we write
it $\nabla \cdot \nabla p = \nabla^2 p$. The Laplacian of a scalar field is also a scalar field and the Laplacian of a vector field is a vector field. The Laplacian of a scalar field writes:

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} \quad (2.4)$$

Finally there is the definition for the Laplacian on a vector field, where the previous equation is applied to each of the three vector components.

$$\nabla^2 \mathbf{u} = \begin{pmatrix} \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2} \\ \frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_y}{\partial z^2} \\ \frac{\partial^2 u_z}{\partial x^2} + \frac{\partial^2 u_z}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} \end{pmatrix} \quad (2.5)$$

In Chapter 5.1 a more practical description of how to approximate these operators using finite differences is discussed.
Chapter 3

Fluid Dynamics

The purpose of this chapter is to give an introduction to some of the physical properties and perceptions that relates to fluid dynamics. We start out with a survey of fluid properties and fluid types. In the final section we arrive at the Navier-Stokes equations, which is based on Newtons second law of motion and conservation of mass and momentum. For a detailed explanation of fluids and their properties we recommend [21].

Fluids are present in many shapes and forms. Typically fluids are loosely defined as the union of liquids and gasses. There exists several substances that are on the border to be fluids or not. When a fluid moves, we say it flows. We give the following definition for fluids:

**Definition 3.1.** Fluids are substances that continuously deforms when subject to shear stress.

Where shear stresses are tangential forces that acts on the fluid surface. One could also think of a fluid as something that cannot resist shear stresses without deformation. This definition includes a large set of fluids that are possible targets for CFD simulations. Among the most common fluid simulations we find water, smoke, fire, explosions, wind, and weather systems. There is no best way to solve all CFD problems, since each type of situation requires different types of configurations and a different set of properties.

### 3.1 Fluid Properties

The behavior of a flow is determined by a set of fluid properties. In the following sections we give a short overview of the properties affecting fluid flows. Detailed descriptions hereof can be found in most textbooks about fluids, for example [21]. The nomenclature in Table 2.1 lists the units of all the following properties.
3.1.1 Density

Density $\rho$, is a common material property, used for many other purposes than just fluid simulations. It is defined as:

**Definition 3.2.** Density $\rho$, is the amount of mass per unit volume.

In the case of smoke simulation, density is a rather small value, compared to water simulation. The higher density, the more force is required to move the same volume of fluid. Which agrees to the assumption that smoke is disturbed more easily than water.

3.1.2 Viscosity

Viscosity is a fluid transport property. The viscosity relates to the stiffness and deformation behavior of the fluid. One might think of viscosity as the internal resistance to deformation.

**Definition 3.3.** Viscosity measures the fluids resistance to shear stress.

As an example a high viscous fluid behaves more like honey than like water. Viscosity has several coefficients, whereas the most common are *kinematic* and *dynamic* viscosity. The kinematic viscosity is simply the dynamic viscosity divided by the fluid density. The dynamic viscosity appears in the derivations of the Navier-Stokes equations, but as the equation simplifies, we arrive at a kinematic definition in Section 3.4. When nothing else is mentioned, we refer to viscosity as the kinematic viscosity.

3.1.3 Pressure

A force applied to a fluid substance, does not immediately affect the entire flow, instead pressure is build up and propagates through the fluid. From a microscopic view, pressure arise from colliding molecules inside the fluids. The definition of pressure is:

**Definition 3.4.** Pressure $p$, is normal force per unit area.

Whereas viscosity relates to the tangential forces, pressure is related to the normal forces. Combined they make up the internal fluid forces as illustrated in Section 3.1.
Chapter 3 3.2 Eulerian vs Lagrangian Methods

\[ F_{\text{total}} = F_{\text{pressure}} + F_{\text{shear}} \]

Figure 3.1: Forces acting on the surface of a fluid. Shear forces are tangential and pressure forces are normal to the surface. Though only the surface is rendered, forces may act the same way inside the entire fluid.

3.1.4 Temperature

Temperature is not explicitly represented in the Navier-Stokes equations. However, temperature still affects the flow behavior, when it varies from the ambient temperature. The temperature difference results in a buoyancy force that makes the fluid go either up or down. In this thesis, we use a buoyancy force for smoke at temperature \( T \) as:

\[ f_{\text{bouyancy}} = -\alpha g (T_0 - T) \] (3.1)

Where \( g \) is the gravitational acceleration vector, \( T_0 \) is the ambient temperature and \( \alpha \) is thermal expansion coefficient. How buoyancy forces are applied to the fluid system is presented later in Section 6.1.1.

3.2 Eulerian vs Lagrangian Methods

The two most widely used approaches in CFD are based on either Eulerian or Lagrangian methods. The difference is whether the fluid parcels are considered relative to an external fixed coordinate system or the system follows the parcels. If the coordinate system is fixed, it is Eulerian, if it has origin at the parcel and follows the parcel, it is Lagrangian. If we consider the substantial derivative of a fluid property \( \phi \), we get:

\[ \frac{d}{dt} \phi(x, t) = \frac{\partial \phi}{\partial t} + \frac{dx}{dt} \frac{\partial \phi}{\partial x} \]

\[ = \frac{\partial \phi}{\partial t} + u \cdot \nabla \phi \] (3.2)

This is also known as the material derivative. For the Lagrangian method, we stated that the coordinate system follows the parcel, hence \( \partial \phi / \partial x \) is zero and the
last term vanishes. For the Eulerian perspective this term remains and is called the advective or sometimes the convective part.

The time consumption of a Lagrangian method depends on the number of parcels simulated, whereas an Eulerian method depends on the size of the computational grid. What method to choose depends on the accuracy, the visual quality, the machine architecture and more. CUDA delivers an extensive set of two and three-dimensional structures and operators. Therefore the Eulerian method fits the GPU programming model very well, using three-dimensional arrays to represent the coordinate system and then store values, such as velocity, temperature, and density at each entry. The Eulerian method also makes it easier to approximate the spatial derivatives using finite difference, which often improves the numerical solution compared to Lagrangian methods.

3.3 Fluid Types

Fluids and fluid flows are classified into different categories dependent on their physical behavior and properties. In the following we describe some of the classifications that are common and useful for CFD models.

3.3.1 Incompressibility

Incompressibility is perhaps the most useful fluid constraint, because it reduces the complexity of the Navier-Stokes equations and thereby the numerical model significantly. Furthermore, it is a rather reasonable constraint to uphold, since compressibility rarely affects any flow with velocities less than subsonic, i.e. a Mach number less than one. The Mach number is often used in CFD to determine whether compressibility should be considered, a threshold of 0.3 is common. In this thesis we will only investigate flows where the velocities are lower. For reference, [9] introduces some methods and techniques for handling compressible flows.

Once a fluid is considered incompressible we can assume that the density is constant over space and time, i.e. the rate of changes are zero:

\[
\frac{\partial \rho}{\partial t} = 0 \tag{3.3}
\]

\[
\nabla \rho = 0 \tag{3.4}
\]

Constant density is the key to simplify the Navier-stokes equations, as we will see in Section 3.4.
3.3.2 Newtonian Fluids

A Newtonian fluid upholds Newton’s law of viscosity. The law states that shear stress is proportional to the angular deformation, with the dynamic viscosity as the scale factor. If the fluid is also incompressible, the following equation describes shear stress:

\[ \tau_{ij} = \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \] (3.5)

Where \( \tau \) is the shear stress tensor and \( \mu \) is the dynamic viscosity. Newtonian fluids can be thought of as any fluid that continuously flattens itself out when placed on a solid surface and only subject to gravity forces. Smoke classifies as Newtonian, which helps when deriving of the momentum conservation.

3.3.3 Inviscid Flow

It is common practise to leave out the diffusive part of the Navier-Stokes equations, if the effect is negligible in relation to what the simulation is used for. In such cases, the fluid is called inviscid, because the diffusive term contains the viscosity property. Such approaches are often seen in GPU based applications, typically in computer game environments, see [27, 26, 34]. The argument for leaving out the diffusive part is, that the contribution to the velocity field is usually small, sometimes even smaller than the errors caused by the numerical model itself. When approximating the convective contribution, incorrect movement is typically introduced and false diffusion appears. False diffusion is recognized as diffuse behavior created by the non-diffusive terms.

In this thesis, we will not use inviscid flows. The reason is that we want the flow to be as accurate as possible, within the limits of the numerical model. As we will see later, we must also diffuse the temperature field, caused by the heat diffusion equation. Thermal diffusion is not as small as smoke diffusion and should not be neglected either.

However, we are aware of the risk, of being inspired by numerical methods, such as [15, 37], that was originally intended for game environments. In the result section we will test for false diffusion and unwanted diffusive behavior.

3.4 The Navier-Stokes Equations

The partial differential equations describing fluid motion was first introduced by Claude-Louis Navier in the 1820s and was 20 years later extended to represent all fluid motions by George Gabriel Stokes. They are therefore known as the Navier-Stokes equations, and are derived from a set of conservation principles. The equations come in several forms, depending on the constraints laid on the system. As
mentioned earlier, we will consider the flow to be Newtonian and incompressible. In this section we present a derivation of the Navier-Stokes equations as they are used in this project. Derivations and explanations of the Navier-Stokes equations are found in most CFD textbooks. For a comprehensive presentation of fluid properties and its physical behavior, we recommend [21, 22]. A short derivation is given here, based on conservation principles with respect to a control volume, as presented in [9] and [21]. It is assumed that the reader is somewhat familiar with physics at graduate level.

### 3.5 Conservation Laws

A conservation law describes the conservation of a given quantity \( \phi \), with respect to time. Two important conservation laws are used to derive the equations of fluid motion, namely mass conservation and momentum conservation. Mass should not leave nor enter the system, therefore we have a constant change equal to zero:

\[
\frac{dm}{dt} = 0
\]  

(3.6)

where \( m \) is mass and \( t \) is time. Momentum is changeable through external forces, the momentum conservation therefore equals the sum of forces:

\[
\frac{d(\rho v)}{dt} = \sum \mathbf{F}
\]  

(3.7)

where \( \rho \) is density and \( \sum \mathbf{F} \) is the sum of forces. When describing a flow, we would like to consider a static spatial region of space, called a control volume (CV). This is contrary to rigid body physics, where the body itself forms the control mass (CM). Therefore, we need an alternative description of the conservation laws, that is given with respect to the control volume. Instead of considering the control mass, we consider the volume that is occupied by the control mass as illustrated in Figure 3.2. The fluid quantities are then described with respect to the control volume, taking into account what enters and leaves the volume. The relation between an extensive quantity \( \varphi \) from a control mass and the corresponding conserved intensive quantity \( \phi \) in the control volume is defined as:

\[
\varphi = \int_{\Omega_{CM}} \rho \phi \, d\Omega
\]  

(3.8)

where \( \rho \) is the density and \( \Omega_{CM} \) represents the volume occupied by the control mass. Equation (3.8) just states that a property of the control mass equals the integral of that same volume times the density. From the equation we see that if \( \phi = 1 \), then the relation for mass is expressed. Likewise, for momentum conservation \( \phi = \mathbf{u} \). To describe the rate of change of the intensive property with respect to the
control volume, we apply the Reynolds transport theorem, see [21] for a detailed derivation. Expressed in terms of $\phi$ it looks as:

$$\frac{d\phi}{dt} = \frac{d}{dt} \int_{\Omega_{CM}} \rho \phi \, d\Omega = \int_{\Omega_{CV}} \frac{d}{dt} \rho \phi \, d\Omega + \int_{S_{CV}} \rho \phi \mathbf{u} \cdot \mathbf{n} \, dS \quad (3.9)$$

where $S_{CV}$ is the surface enclosing the control volume and $\mathbf{n}$ is the outward unit vector orthogonal to the surface. The equations states that the rate of change of $\phi$ from the control mass, equals the change of the same property in the control volume, plus the net flux through the surface. It makes good intuitive sense that the change of a quantity inside a static volume are related to the amount that flows in minus the amount that flows out. The situation is illustrated in Figure 3.3.

We are now able to express the conservation laws with respect to a control volume and the surrounding surface. However, we would still like to express the conservations entirely using the control volume. Therefore we finally apply Gauss’ divergence theorem, replacing the surface integral from (3.9) with a volume inte-
Chapter 3  3.5 Conservation Laws

d \int_{\Omega_{CM}} \rho \phi \, d\Omega = \int_{\Omega_{CV}} \frac{d}{dt} \rho \phi + \nabla \cdot (\rho \phi \mathbf{u}) \, d\Omega \quad (3.10)

The right hand side is now the basis formula for the mass and momentum conservation laws. The equation states that the rate of change of the extensive quantity \( \phi \) equals the rate of change of the intensive quantity inside the control volume plus the velocity divergence.

### 3.5.1 Mass Conservation, The Continuity Equation

Inserting the quantity for mass, \( \phi = 1 \), into equation (3.10), we get the rate of mass change, which we know from (3.6) must be zero. If we let the control volume become infinitely small the integral vanishes and we end up with the mass conservation, also known as the **continuity** equation:

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 
\quad (3.11)
\]

Since the fluid is incompressible as described earlier, the density must be constant and non-zero, also stated in (3.3). Incompressibility reduces the continuity equation to:

\[
\nabla \cdot \mathbf{u} = 0 
\quad (3.12)
\]

### 3.5.2 Momentum Conservation

The continuity equation is a constraint, forcing the divergence of the velocity field of a flow to be zero. It does not tell anything about how the flow develops, for that we need the momentum conservation. Similarly to mass conservation we insert \( \phi = \mathbf{u} \) into (3.10) and use the definition for momentum conservation from (3.7). \( \mathbf{u} \) is a three component vector, hence there will be three equations. In the following we only show the derivation using the x-component, though the same holds for the y- and z-components. We get:

\[
\int_{\Omega_{CV}} \frac{d}{dt} \rho u^x + \nabla \cdot (\rho u^x \mathbf{u}) \, d\Omega = \sum F^x 
\quad (3.13)
\]

Equation (3.13) does not provide much useful information, since the sum of forces is unknown. It is necessary to investigate the forces acting on a flow. The forces are categorized in two classes, body forces and surface forces. The momentum conservation becomes:

\[
\int_{\Omega_{CV}} \frac{d}{dt} \rho u^x + \nabla \cdot \rho u^x \mathbf{u} \, d\Omega = \int_{\Omega_{CV}} F_B^x \, d\Omega + \int_{S_{CV}} F_S^x \, dS 
\quad (3.14)
\]
The body forces act on the entire volume, they are usually forces such as gravity and buoyancy. That is also the two types of body forces supported by our CFD solver. In section 6.1.1 we describe how these forces are calculated and applied to the velocity field.

Before we investigate the surface forces in details, we can simplify the second term on the left hand side, using product-rule for differentiation, the continuity equation and the fact that $\rho$ is constant to get:

$$
\nabla \cdot \rho u^x u = u \cdot \nabla (\rho u^x) + \rho u^x \nabla \cdot u
$$

$$= \rho (u \cdot \nabla) u^x$$

(3.15)

To summarize, we can now insert (3.15) into (3.14) and obtain:

$$
\int_{\Omega} \frac{d}{dt} \rho u^x d\Omega + \rho (u \cdot \nabla) u^x d\Omega = \int_{\Omega} F_B^x d\Omega + \int_{S} F_S^x dS
$$

(3.16)

Collecting all three vector components back into one equation enables us to rewrite equation (3.16) to vector form:

$$
\int_{\Omega} \frac{d}{dt} \rho u + \rho (u \cdot \nabla) u d\Omega = \int_{\Omega} F_B d\Omega + \int_{S} F_S dS
$$

(3.17)

The surface forces described by the vector $F_S$, is a combination of shear stress and pressure. To make the combination more distinct, we can express $F_S$ as a matrix vector product that helps define the surface forces.

$$
F_S = \Gamma \cdot n
$$

(3.18)

Where $n$ is the outward pointing normal vector and $\Gamma$ is a 3x3 matrix. Since $n$ is purely geometrical, the matrix $\Gamma$ must contain all the shear stress and pressure contribution. Inserting the new definition for $F_S$ into the right hand side of (3.17) and applying Gauss’ theorem gives:

$$
\int_{\Omega} F_B d\Omega + \int_{S} \Gamma \cdot n dS = \int_{\Omega} F_B + \nabla \cdot \Gamma d\Omega
$$

(3.19)

We now use the assumption that the fluid is Newtonian. When the fluid is Newtonian, we can determine the structure and content of $\Gamma$. When the fluid is incompressible and Newtonian, the shear stress contribution to $\Gamma$ is described by (3.5). Pressure also contributes to the content of $\Gamma$. The description and derivation of $\Gamma$ is quite comprehensive and is not of much interest for the purpose of the thesis. Therefore we will leave out this part and instead refer to [21] for a full description. The divergence of $\Gamma$ yields two terms, called the diffusive and the pressure terms.

$$
\int_{\Omega} \mu \nabla^2 u - \nabla p + F_B d\Omega = \int_{\Omega} \mu \nabla^2 u - \nabla p + F_B d\Omega
$$

(3.20)
Where $\mu$ is the kinematic viscosity and $p$ is the pressure. Finally we combine the derivation of the forces and the momentum conservation in (3.17) and get the Navier-Stokes equation in integral form:

$$\int_{\Omega_{CV}} \frac{d}{dt} \rho u + \rho (u \cdot \nabla) u \, d\Omega = \int_{\Omega_{CV}} \mu \nabla^2 u - \nabla p + F_B \, d\Omega$$  \hspace{1cm} (3.21)

Under the assumption that we consider an arbitrary small control volume, which implies that the integrand becomes zero, and dividing with the constant density $\rho$, we end up with the Navier-Stokes equation in differential form:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla) u = \nu \nabla^2 u - \frac{1}{\rho} \nabla p + f$$  \hspace{1cm} (3.22)

This concludes our derivation of the Navier-Stokes equations. We note that (3.22) is a non-linear partial differential equations, with respect to time and space, where the space derivatives are implicit represented in the $\nabla$ operators. We also note that the equation consists of five individual terms. They each describe a particular part of the physical characteristics of the fluid flow, further discussed in Section 6.1. The momentum equation is used to continuously solve for new velocities, i.e. calculate $du/dt$ with a time step $\Delta t$, and apply the changes to the velocity field.

### 3.5.3 Reynolds Numbers

If we make the Navier-Stokes equations dimensionless, that is, we divide every term with a characteristic value of the same unit, we end up with a couple of numbers, known as the Reynolds, Froude, and Strouhal numbers. The derivation of these numbers can be found in [9]. From an engineering point of view, the Reynolds number is often interesting, because it describes the flow state (turbulent or not) and because it makes flows comparable at different scales. The Reynolds number is defined as:

$$Re = \frac{\rho u_0 L_0}{\mu} = \frac{u_0 L_0}{\nu}$$  \hspace{1cm} (3.23)

Where $u_0$ is the characteristic velocity and $L_0$ is the characteristic length. These values are determined by the specific configuration. Usually $u_0$ will be the average velocity and $L_0$ the geometric diameter of the room or pipe.

In Chapter 8 our results will be compared to some classical test cases where results have been presented in various publications. The Reynolds number is used here to ensure comparability between different solvers.
Chapter 4

Parallel Computing using CUDA

In the past few years, methods originally targeted the CPU, have been considerably improved using the powers of alternative architectures, such as the Cell processor or the GPU. The GPU Gems book series[8, 32, 27] is a good collection of articles that emphasizes that the GPU is not only meant for rendering purposes. However, most general purpose GPU (GPGPU) methods are based on traditional shader programming techniques, which is often not intuitive. CUDA was designed by NVIDIA to remove this abstraction, enabling programmers to use traditional C code, without having to include any render-specific commands.

This section gives an overview of the CUDA architecture and presents some of the extended C syntax used to implement our CFD solver. The purpose is not to write a comprehensive CUDA user guide, but merely to give an insight into the CUDA aspects that have been touched throughout this project. For the purpose of a user guide, we recommend [29]. Since CUDA documentation limits almost only to [29] and [30], our introduction is heavily based on these articles.

4.1 Programming Model

CUDA distinguishes between the *host* and the *device*. The GPU works as a physically separate coprocessor to the host, running the main C program. The CPU is therefore called the host and the GPU the device. Since the host and device are physically separated, they maintain their own memory regions, likewise called the host memory and device memory. Allocation of the device memory goes through system calls from the host side to the CUDA API. The host and device relationship is depicted in Figure 4.1. Methods executed on the device are called kernels. Kernels are defined using the keyword `__global__` prior to the method declaration. Kernels are executed in parallel on the device, using the `<<<...>>>` syntax in the host code. In parallel means that multiple threads execute the same program at the
same time on the GPU. The hardware specifications, such as the number of processors, ALUs, memory capacity, and bandwidth, depends on the specific model. We refer to [29] for details and also for description of the architectural configuration.

A simple example adding two vectors \( a \) and \( b \) of size \( N \) are given in Listing 4.1.

```c
// The device code
__global__ void vectorAdd(float *a, float *b, float *c)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

// The host code
void main()
{
    // Create and allocate N, a, b and c here
    // Invoke kernel
    <<<1, N>>>vectorAdd(a, b, c);
}
```

Listing 4.1: Device and host code. The kernel is defined using the \__global__\ keyword. The host invokes the kernel on \( N \) threads, using the \<<<...>>>\ syntax.

The example leaves out the initialization of all variables, we will describe how to do this later. Notice that the kernel reads an index variable from threadIdx. These built-in variables are very important, they are used to distinguish the threads from one another. The following section describes how kernels use the built-in variables and how they are properly invoked using the \<<<...>>>\ syntax.
4.1.1 Threads, Blocks and Grids

Threads are organized in blocks. A block is a one, two or three dimensional structure of threads. In the vector addition example in Listing 4.1, a one dimensional block would be the obvious choice, but for matrix addition, a two dimensional block would be more intuitive. What dimension to choose, is highly dependent on the specific problem. Regardless of dimensions, a block is limited to a total of 512 threads on all current GPUs, see [29] for details. If a problem domain becomes too big, we must use several blocks. To solve this, blocks are again organized in grids. A grid is restricted to one or two dimensions. The number of blocks in a grid is limited to more than 4 billions, which is more than enough for most purposes. However, limiting grids to two dimensions has shown to be problematic in a three dimensional CFD solver situation, as will be discussed in Section 7.1. Figure 4.2 illustrates an example of the connection between blocks and grids.

![Figure 4.2: A two dimensional grid of eight blocks. Each block is also two dimensional, consisting of 12 threads, yielding a total of 96 threads.](image)

Grids and blocks are the two arguments that goes inside the <<<...>> tag when executing a kernel from the host. Both the grids and blocks are of type `dim3`, a three dimensional integer vector, though the third dimension is ignored for grids.
4.1 Programming Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadIdx</td>
<td>uint3</td>
<td>Unsigned integers identifying the thread in the block.</td>
</tr>
<tr>
<td>blockIdx</td>
<td>uint3</td>
<td>Unsigned integers identifying the block in the grid.</td>
</tr>
<tr>
<td>blockDim</td>
<td>dim3</td>
<td>Contains the sizes of the block dimension.</td>
</tr>
<tr>
<td>gridDim</td>
<td>dim3</td>
<td>Contains the sizes of the grid dimension.</td>
</tr>
</tbody>
</table>

Table 4.1: Built-in variables to identify the threads and blocks.

Once a kernel is invoked, four built-in variables are available to identify the threads and blocks. The variables are listed in Table 4.1. Let’s summarize using another simple example. The example in Listing 4.2 adds two matrices, allocated in linear memory.

```
// The device code
__global__ void vectorAdd(float *a, float *b, float *c, int I, int J)
{
  int i = threadIdx.x + blockDim.x*threadIdx.y;
  if (i < I*J)
    c[i] = a[i] + b[i];
}

// The host code
void main()
{
  int I;
  int J;
  // Create and allocate I, J, a, b and c here
  // Create block and grid
  Dim3 block(16, 16);
  Dim3 grid((I + block.x - 1)/block.x, (J + block.y - 1)/block.y);
  // Invoke kernel
  <<<grid, block>>>add(a, b, c, I, J);
}
```

Listing 4.2: The host creates a block of threads and a grid before invoking the kernel. The kernel uses the built-in variables to calculate the correct indices.

4.1.2 Device Memory

Device memory are divided into three main spaces, local memory, shared memory and global memory. Figure 4.3 illustrates the memory setup. Only the global memory are persistent between kernel calls, so it is the obvious place to store the vector fields that are needed by the CFD solver. Unfortunately, reading and writing from global memory is slow, compared to data access from local and shared memory. [29] has a chapter on accessing device data in an optimal manner, but it is not a trivial task to uphold all constraints. If possible, it is recommended to read values from global memory and store them in shared memory for future use. This is often useful when each thread calculates results based on multiple neighbor values.
However, such access to shared memory requires the threads to be synchronized and extra care must be taken. Rewriting the kernels is also required, which makes the code less intuitive. It is therefore often a tradeoff between simplicity and efficiency to optimize memory accesses. In Section 7.1 we describe how some of the kernels were optimized using shared memory in place of global memory.

Global memory is allocated on the device through calls to the CUDA API. In order to allocate memory from the host, one must specify the size of the space needed and a pointer that will point to the memory after allocation. The allocation method to use, depends on the type of memory needed. CUDA delivers structures for both one, two, and three dimensional arrays. In this project we have only used three dimensional CUDA arrays, Listing 4.3 gives an example of allocating such an array.
4.1 Programming Model

CUDA only supports texture copies of float4 values, which is the reason that we do not use float3 for the velocity field.

The `cudaPitchedPtr` structure gathers information of the memory allocation. CUDA automatically allocates device memory in sizes that aligns to the GPU architecture. Therefore, array indexing must be handled in a rather tedious manner.

The important thing to notice is that one might expect that the index should be calculated as:

```c
size_t idx = threadIdx.x + threadIdx.y*blockDim.x + threadIdx.z*blockDim.y;
```

But since the width stored in `blockDim.x` does not necessarily equal the actual width that was allocated, the pitch value must be used instead.

This pitfall caused quite some headache, both because it is counterintuitive and because it requires intensively use and passing of pitch values.
4.1.3 Textures

Though textures are mostly meaningful for rendering purposes, CUDA still provides methods for allocating and fetching textures values. Textures have a couple of nice properties that sometimes make them useful for GPGPU programs. Contrary to arrays indexing, values from a texture are fetched using floating point coordinates. Values fetched in between integer coordinates are automatically interpolated by the hardware. This means that we can get the hardware to do the interpolation if the array is first copied to a texture. Furthermore, the addressing mode of a texture can be set to clamp or wrap coordinates to the texture dimensions. Hence, indexing out of range is not an issue.

A texture is actually just a reference that defines which part of memory to fetch from. A texture is read-only, so if one wants to update a texture, it must be done through copy methods provided by the CUDA API. Figure 4.4 illustrates how a texture is updated from a CUDA array.

![Figure 4.4: Updating textures goes through the CUDA method cudaMemcpy3D.](image)

One last thing to be aware of is, that fetching from a texture is translated by one half, compared to array indexing. For example, the first value in a two dimensional array would be indexed with the integer 0, whereas the same value from a referenced texture is fetched with the coordinates (0.5, 0.5).
4.2 Numerical Precision

Solving the Navier-Stokes equations for a three dimensional velocity field, obviously requires structures that easily adapt to three dimensions. The CUDA API delivers built-in vector types to handle vector operations in both integer and floating point precision. Unfortunately, only GPUs of compute capability 1.3 and higher support double precision arithmetic. Currently this only includes the very newest and most expensive GPUs. Studies have been done on enhancing GPU precision. [1] presents a correction algorithm, that enhances the precision computed on the GPU with an iterative correction algorithm running in double precision on the CPU. Since double precision compatible GPUs have entered the market and prices usually drop fast, we will not spend time on implementing such correction algorithms.
Chapter 5

Discretization

The Navier-Stokes equations give a continuous description of the fluid flow, which is only analytic solvable for a sparse set of rather simple situations. Instead we must make a discrete approximation of time and space. Spatial discretization partitions the area of interest into a finite numerical representation. As described in Section 3.2 Eulerian methods fit the GPU architecture well, representing the flow with respect to a static grid. We will also use the term ‘mesh’ to avoid misconception with a CUDA grid. The form and resolution of the grid are usually determined by the accuracy needed and the complexity of the problems to be solved. The simplest grid type is a uniform Cartesian grid. Cartesian means that the axes of the grid are aligned with the world axes. Uniformity tends to be defined differently in the literature. In this thesis we use the following definition:

**Definition 5.1.** All cells $c$, in a uniform grid $\mathcal{G}$, are of the same cubic size and fulfill:

$$\Delta x_c = \Delta y_c = \Delta z_c, \quad \forall c \in \mathcal{G}$$

Where $\Delta x_c, \Delta y_c, \Delta z_c$ is the width, height and depth of the cell. [15], [26], and [34] use such uniform Cartesian grids, which simplify some of the finite difference equations presented in Section 5.1. However, this will not suffice for our purpose. Instead we will use a regular Cartesian grid defined as:

**Definition 5.2.** For all pair of cells $c1, c2$, in a regular grid $\mathcal{G}$, the following relation hold.

$$\begin{align*}
\Delta x_{c1} &= \Delta x_{c2} \\
\Delta y_{c1} &= \Delta y_{c2} \\
\Delta z_{c1} &= \Delta z_{c2}, \quad \forall c1, c2 \in \mathcal{G}
\end{align*}$$
Regular grids are sometimes also referred to as uniform, but we will not do so here. The advantage of using non-uniform grids is that both speed and accuracy are usually obtained when using a fine resolution in the direction the fluid is moving, and a coarser resolution in the less active directions. The difference between a uniform and a regular grid is illustrated in figure 5.1.

Other more advanced grids exist, such as rectilinear and curvilinear, often used when the interior of the simulation is known in advance, in which case the grid can be shaped to fit the boundaries. Such grids are both non-uniform and non-regular. A regular grid fits the GPU architecture well and it still delivers the flexibility to divide each grid dimension in different sizes. Grids in FDS is defined the same way, so it will let us reuse the same input description format. Once settled on a regular grid, we must also determine how to index the values stored at the cells. The cells store the values for the velocity field, pressure field, force field etc. There are two primary approaches that have been intensively used in literature, collocated and staggered grids. The difference lies in how the velocity vectors are tied to the grid. In a collocated grid the vectors are located in the center of the cells, whereas for a staggered grid, the vectors are located componentwise at the cell faces. A two dimensional example is illustrated in Figure 5.2. Obviously the collocated grid are the most intuitive and the easiest to implement. The staggered grid has the essential advantage that boundaries can be represented explicit, since velocities are stored exactly where the boundaries will be. Central differences in the center of the cells may also be calculated directly from the values stored at the cell faces, which minimizes numerical dissipation. We have chosen the collocated grid because of its simplicity and because it fits the CUDA architecture well. If the solution turns out to suffer from dissipation, switching to a staggered grid might serve as an improvement. However, it has not been tested due to time constraints and will be left as future work.

To ease notation, we use subscripts to indicate the location from which a value belongs. Let \( P \) be a function that returns the value \( \phi \) from a cell, then we have:

\[
\phi_{i,j,k} = P(c_{i,j,k}), \quad \forall c_{i,j,k} \in \mathcal{G}_{I,J,K}
\]
Chapter 5

5.1 Finite Difference

Figure 5.2: **Left:** Collocated grid with velocities represented in the center of the cells. **Right:** Staggered grid with the velocities represented on the faces between the cells.

Where \( c_{i,j,k} \) is the cell at location \((i,j,k)\), from which the value \( \phi_{i,j,k} \) belongs. When \( i, j, \) or \( k \) are not integers, \( \phi \) is interpolated from the nearest discrete values. In three dimensions this is a trilinear interpolation based on eight values.

5.1 Finite Difference

The collocated grid enables us to give discrete definitions of the spatial differential operator presented in Section 2.2 using finite differences. Finite difference is a numerical approach to estimate derivatives using the discrete values in the grid. For a differentiable function \( f \), we can express the function value at any point \( x \) close to a fixed point \( x_i \) using the Taylor expansion:

\[
f(x) = f(x_i) + (x - x_i) \frac{\partial f(x_i)}{\partial x} + \frac{(x - x_i)^2}{2!} \frac{\partial^2 f(x_i)}{\partial x^2} + \mathcal{O}((x - x_i)^3) \tag{5.1}
\]

Where \( \mathcal{O}((x - x_i)^3) \) represent the higher order terms. Now, if \( x \) is exchanged with one of the discrete adjacent points, we can rearrange and get an expression for the first derivative at \( x_i \). If both of the adjacent points are used \((x_{i-1} \) and \( x_{i+1} \)) and the higher order terms are discarded, we obtain an approximation known as central difference, we refer to [9] for a more detail derivation.

\[
\frac{\partial f(x_i)}{\partial x} \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{5.2}
\]

Such an approximation of the first derivative is first order accurate. More precise approximations exists, such as polynomial fitting, but they require information from multiple points. We have chosen to use central difference for the spatial derivatives. Higher order approaches would not be worth the extra cost, because the time derivatives from the fractional step method presented in the next section, will not be more accurate. Besides, the neglected error term scales with the cell sizes, i.e. to enhance the results we can increase the resolution of the computational
The same approximations hold in three dimensions. Using central difference on the gradient of a scalar field $p$, as introduced in Equation (2.2) in Section 2.2, yields:

$$\nabla p = \left( \frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z} \right)^T$$  \hspace{1cm} (5.3)

$$\approx \left( \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\Delta x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\Delta y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\Delta z} \right)^T$$  \hspace{1cm} (5.4)

Likewise the divergence from (2.3) of a vector field can be defined in terms of central difference as:

$$\nabla \cdot u = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$  \hspace{1cm} (5.5)

$$\approx \frac{u_{i+1,j,k}^x - u_{i-1,j,k}^x}{2\Delta x} + \frac{u_{i,j+1,k}^y - u_{i,j-1,k}^y}{2\Delta y} + \frac{u_{i,j,k+1}^z - u_{i,j,k-1}^z}{2\Delta z}$$  \hspace{1cm} (5.6)

Finally when we apply the central difference two times to the scalar field $p$, we get the second derivative from equation (2.4):

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}$$  \hspace{1cm} (5.7)

$$\approx \frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{(\Delta x)^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{(\Delta y)^2} + \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{(\Delta z)^2}$$  \hspace{1cm} (5.8)

For the velocity field the Laplacian is applied on each of the three vector components and gives:

$$\nabla^2 u = \begin{pmatrix}
\frac{u_{i+1,j,k}^x - 2u_{i,j,k}^x + u_{i-1,j,k}^x}{(\Delta x)^2} & \frac{u_{i,j+1,k}^y - 2u_{i,j,k}^y + u_{i,j-1,k}^y}{(\Delta y)^2} & \frac{u_{i,j,k+1}^z - 2u_{i,j,k}^z + u_{i,j,k-1}^z}{(\Delta z)^2} \\
\frac{u_{i+1,j,k}^y - 2u_{i,j,k}^y + u_{i-1,j,k}^y}{(\Delta x)^2} & \frac{u_{i,j+1,k}^z - 2u_{i,j,k}^z + u_{i,j-1,k}^z}{(\Delta y)^2} & \frac{u_{i,j,k+1}^x - 2u_{i,j,k}^x + u_{i,j,k-1}^x}{(\Delta z)^2} \\
\frac{u_{i+1,j,k}^z - 2u_{i,j,k}^z + u_{i-1,j,k}^z}{(\Delta x)^2} & \frac{u_{i,j+1,k}^x - 2u_{i,j,k}^x + u_{i,j-1,k}^x}{(\Delta y)^2} & \frac{u_{i,j,k+1}^y - 2u_{i,j,k}^y + u_{i,j,k-1}^y}{(\Delta z)^2}
\end{pmatrix}$$  \hspace{1cm} (5.9)

How the time derivative is handled will be presented in the next section, in which we presents a fractional step method to solve the unsteady Navier-Stokes equations.
Chapter 6

The Fractional Step Method

A wide variety of methods for solving CFD problems have been proposed during the years. In this thesis we have been inspired by the fractional step approach as presented by Stam and Harris in [36, 15] and later improved by Crane in [18]. The coupling between the divergence free velocity field and the pressure field, is controlled through a projection, similar to the one presented in [15]. Both [18] and [15] have the advantage that they have been implemented on the GPU using shader programming. Motivated by intermediate results and conversations with Rambøll, we decided to improve the linear solver. To enhance the convergence rate, a multigrid approach was applied to the Jacobi solver for the pressure Poisson equation, presented in Section 6.1.4. The multigrid solver fits the parallel architecture of the GPU well and enables the Jacobi solver, as presented in [15], to be reused for the relaxation steps.

6.1 Navier-Stokes in Fractions

A common approach to solve partial differential equations is to split them into smaller parts, and solve them independently. In Equation (3.7) we showed that the Navier-Stokes momentum equation consists of four parts, repeated here.

$$\frac{\partial \mathbf{u}}{\partial t} = -\left(\mathbf{u} \cdot \nabla\right) \mathbf{u} + \nu \nabla^2 \mathbf{u} - \frac{1}{\rho} \nabla p + \mathbf{f} \quad \text{(6.1)}$$

For each discrete time step, we split the equation and update the velocity field in four steps. Each fraction describes an acceleration to the velocity field. Updating the velocity field from an already known velocity field $\mathbf{u}_0$ can be sketched as:

\[ \mathbf{u}_0 \xrightarrow{\text{external forces}} \mathbf{u}_1 \xrightarrow{\text{advection}} \mathbf{u}_2 \xrightarrow{\text{diffusion}} \mathbf{u}_3 \xrightarrow{\text{pressure}} \mathbf{u}_4 \]
Chapter 6 6.1 Navier-Stokes in Fractions

Our solution uses this update order. Though, in general it is commutative, i.e. it is not restricted to any particular order. Writing the same fractional steps in differential form we get:

\[
\begin{align*}
\frac{\partial u_1}{\partial t} &= f \\
\frac{\partial u_2}{\partial t} &= (u_1 \cdot \nabla)u_1 \\
\frac{\partial u_3}{\partial t} &= \nu \nabla^2 u_3 \\
\frac{\partial u_4}{\partial t} &= \frac{1}{\rho} \nabla p
\end{align*}
\]  
(6.2) (6.3) (6.4) (6.5)

The last fraction is actually not solved exactly as listed here, but as a consequence of satisfying the continuity equation using a Helmholtz-Hodge decomposition. This will be covered in detail in Section 6.1.4. The following sections describes the approach we have used to solve each of the four differential equations.

6.1.1 External Forces

The addition of external forces is probably the simplest term in the Navier-Stokes equations. Updating the velocity field from a given force field \( f \), we use the explicit Euler scheme:

\[
\vec{u} = u + \Delta t f
\]  
(6.6)

Where \( \vec{u} \) is the updated velocity field. For this update to make sense, the force field must also be updated repeatedly to contain the forces acting on the fluid. Remember that \( f \) is actually force per density, but as density is constant, the difference is only a constant scaling. As mentioned earlier, the two types of external forces acting on the fluid are gravity and thermal buoyancy.

From Newton’s Second Law of Motion, we know that the gravitational force equals mass times acceleration. In vector form that is:

\[
f_{\text{gravity}} = mg
\]  
(6.7)

Where \( m \) is mass and \( g \) is the gravitational acceleration, a vector pointing the negative vertical direction.

Buoyancy forces work in the opposite direction than gravity. Buoyancy is what cause airplanes to lift from the ground and hot air to rise above cold air. The latter is called thermal buoyancy. We use an approximation as it is presented in [15]:

\[
f_{\text{buoyancy}} = -\alpha g (T - T_0)
\]  
(6.8)

Where \( \alpha \) is a constant coefficient that controls the effect of the buoyancy forces. \( T_0 \) is the ambient temperature in the room and \( T \) is the local temperature of the
fluid. It is not difficult to see from the equation, that the effect of thermal buoyancy also scales linearly with the difference between the ambient and local temperature. Therefore, it is convenient to gather the two forces into one equation as:

\[ f = g m - \alpha g (T - T_0) \]  

(6.9)

This is a widely used approximation, known as the Boussinesq approximation. Boussinesq approximations are good when the temperatures do not differ much, but as stated in [20], it is not a recommended approach when simulating fire-smoke interactions, though it is fast and simple. Since we do not simulate the fire itself, but only the smoke, we expect it to be a reasonable approximation.

Updating the force field using (6.9), requires knowledge of masses and temperatures inside the simulation domain. Two additional scalar fields is therefore required, and must also be simulated according to the velocity field. Section 6.3 discuss how to handle these fields.

To summarize, (6.9) describes how to update the force field, which is subsequently applied to (6.6) to calculate the total effect of external forces to the velocity field.

6.1.2 Advection

Advection describes the motion the fluid causes on itself and other passive quantities. In literature, advection is also referred to as convection - they mean the same. Think of advection as the motion that would be created if a passive material was injected into a flow, like ink in water for example. Advection is non-linear and described by the term:

\[ \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} \]  

(6.10)

An obvious approach to solve the advection step is to use an explicit Euler scheme, multiplying both sides with the time step to obtain:

\[ \Delta \tilde{\mathbf{u}} = \mathbf{u} - \Delta t (\mathbf{u} \cdot \nabla)\mathbf{u} \]  

(6.11)

Using finite difference as presented earlier, this will reduce to a rather simple advection scheme. Unfortunately explicit methods tends to be unstable if the time steps are too big. In fact, to ensure that the solution does not blow up, the time step \( \Delta t \), multiplied the largest velocity \( \mathbf{u}_{\text{max}} \), should never give a velocity with a magnitude larger than the size of the cells. This restriction is known as CFL conditioning, see [5] for details.

Instead we consider the implicit Semi-Lagrangian method, first proposed by Stam in [36]. This method is unconditionally stable, regardless of the time steps. The idea is to trace velocities back in time to find the new velocities to advect. The position that is traced \( \Delta t \) back in time from the initial position \( x \), we define as
\( p(x, -\Delta t) \). Using Semi-Lagrangian advection the velocity update at position \( x \) becomes:

\[
\mathbf{u}(x) = \mathbf{u}(p(x, -\Delta t))
\]

(6.12)

Obviously the back trace might end in-between the cell centers, in such case the value \( p(x, -\Delta t) \) must be interpolated between neighbour cell values. If the values are fetched from a texture, the GPU hardware will interpolate them automatically. The method is illustrated in Figure 6.1. Since new values are always gathered from the direction the flow is moving, the scheme is also called an upwind scheme. An additional good property of the Semi-Lagrangian method is, that it fits a parallelized architecture well, because each thread stores values in independent cells, avoiding write collisions.

Despite its good qualities, the Semi-Lagrangian method also has some less attractive properties. New values are always an average of previous values. This is what makes the method stable, but it also introduces intense smoothing, removing many details. Thus, introducing false diffusion. False diffusion is characterized as any diffusive behavior that is not caused by the diffusive term. [25] claims that a MacCormack advection scheme could be extended from the Semi-lagrangian method, as shown in [15], in order to gain second order accuracy. The idea is basically to add a correction step using an explicit Euler scheme. Since the spatial derivatives are only first order accurate the improvement will only be minor. Thus, we will leave such improvements for future work.

Figure 6.1: Unconditionally stable Semi-Lagrangian advection of the velocity field \( \mathbf{u} \). The new velocity at position \( \mathbf{x} \) will be \( \tilde{\mathbf{u}} \)
6.1.3 Diffusion

Flow resistance, which is tied to the fluid viscosity as described earlier, gives the flow diffusive behavior. The diffusive term describes the acceleration causing interaction between adjacent cells. It is defined as:

\[
\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u} \tag{6.13}
\]

To solve the diffusion equation, we once again prefer an implicit approach to ensure stable behavior. This implicit formulation was inspired by Stam in [36] and writes:

\[
(I - \Delta t \nu \nabla^2) \tilde{\mathbf{u}} = \mathbf{u} \tag{6.14}
\]

Where \( I \) is the identity matrix. This is a system of linear equations, known as a Poisson equation. So, if we have a solver for linear systems and a known velocity \( \mathbf{u} \), we can solve for the new velocity field \( \tilde{\mathbf{u}} \). In a moment we will see, that a Poisson solver is also needed for the pressure term. Thus, we will defer the description of how to solve these systems to Section 6.2.

6.1.4 Pressure

When fluid parcels push each other at a molecular level, pressure builds up and propagates through the entire flow. Areas with high pressure will move towards low pressure areas. The acceleration caused by pressure is described by the pressure term:

\[
\frac{\partial \mathbf{u}}{\partial t} = \frac{1}{\rho} \nabla p \tag{6.15}
\]

The way we solve the pressure term is somewhat different than the way we solved the previous terms. Until now we have not been concerned with the continuity equation so the velocity field obtained so far may very well be divergent. For incompressible flows we can use the pressure field to force continuity, i.e. create a pressure projection of the divergent velocity field into a divergence-free field. If we write out (6.15) using backward difference we get:

\[
\frac{\mathbf{u} - \tilde{\mathbf{u}}}{\Delta t} = \frac{1}{\rho} \nabla p \tag{6.16}
\]

Where \( \tilde{\mathbf{u}} \) will be the final divergent-free velocity field and \( \mathbf{u} \) is the velocity field obtained after applying forces, advection and diffusion. From (6.16) we can write an expression for \( \tilde{\mathbf{u}} \) as:

\[
\tilde{\mathbf{u}} = \mathbf{u} - \frac{\Delta t}{\rho} \nabla p \tag{6.17}
\]

The problem is now to find a \( p \) such that \( \tilde{\mathbf{u}} \) becomes divergence-free when we subtract the pressure term from \( \mathbf{u} \). If we apply the divergence operator to both
sides of (6.17) we know that the left hand side will be zero by the definition of the continuity equation:

\[ \nabla \cdot \tilde{u} = \nabla \cdot u - \frac{\Delta t}{\rho} \nabla^2 p \quad (6.18) \]

\[ \nabla^2 p = \frac{\rho}{\Delta t} (\nabla \cdot u) \quad (6.19) \]

Only \( p \) is unknown now, and it is again a Poisson equation. As before, we will defer the discussion of how to solve this equation to Section 6.2. Once (6.19) is solved for \( p \), we insert it back into (6.17) and calculate the gradient. The gradient is then used for the projection that will make \( \tilde{u} \) divergence-free. The two density fractions in (6.19) and (6.17) cancel out each other, simplifying the equations a bit. Collecting the pieces in a scheme gives:

1. Apply external forces, advection, and diffusion to arrive at \( u \)
2. Solve \( \nabla^2 p = \nabla \cdot u \) for \( p \)
3. Calculate the projection \( \tilde{u} = u - \nabla p \) to make \( \tilde{u} \) divergence-free.

This ends our presentation of the fractional step method used to update the velocity field. The only thing that remains, is how to handle boundaries, updating the scalar fields, and how to solve the linear systems. All three subjects will be handled in the following sections.

### 6.2 Solving Linear Systems

The fractional step method describes above introduces two linear Poisson equations to be solved, repeated here:

\[ (I - \nu \delta t \nabla^2) \tilde{u} = u \quad (6.20) \]

\[ \nabla^2 p = \nabla \cdot u \quad (6.21) \]

Though they might not seem alike, they are in fact both Poisson equations. A Poisson equation is a partial differential equation, in Euclidian space it is of the form \( \nabla^2 \phi = f \). Since both equations are Poisson equations, it is possible to construct a solver such that the same routine can be used to solve both equations. Taking one equation out of (6.21) and writing it out using central difference, de-
scribed in Section 5.1 we get:

\[
\begin{align*}
& p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k} \\
& + p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k} \\
& + p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1} \\
& = \frac{u_{i+1,j,k}^x - u_{i-1,j,k}^x}{2\Delta x} + \frac{u_{i,j+1,k}^y - u_{i,j-1,k}^y}{2\Delta y} + \frac{u_{i,j,k+1}^z - u_{i,j,k-1}^z}{2\Delta z}
\end{align*}
\]  

(6.22)

This is how one equation looks like, but there will be as many equations as there are cells in the grid. I.e. the total number of equations equals the resolution of the grid, \( I \times J \times K \), which we denote \( N \). Boundary cells is an exception to this, we discuss boundaries in Section 6.4. The right hand side of (6.22) is constant when solving for \( p \), so to shorten notation there is no need to write this side out explicitly.

If we rearrange the equations so that the constant coefficients from the Laplacian operator appear separately we get:

\[
\begin{align*}
& \frac{1}{(\Delta x)^2} (p_{i+1,j,k} + p_{i-1,j,k}) \\
& + \frac{1}{(\Delta y)^2} (p_{i,j+1,k} + p_{i,j-1,k}) \\
& + \frac{1}{(\Delta z)^2} (p_{i,j,k+1} + p_{i,j,k-1}) \\
& - \left( \frac{2}{(\Delta x)^2} + \frac{2}{(\Delta y)^2} + \frac{2}{(\Delta z)^2} \right) p_{i,j,k} \\
& = \nabla \cdot \mathbf{u}
\end{align*}
\]  

(6.23)

Each equation in the linear system, involves seven values from the pressure field, as illustrated in Figure 6.2, we call it a molecule and its structure is bound to the difference scheme. Higher order difference schemes would involve more adjacent values. The entire matrix system obtained from (6.21) is illustrated in Figure 6.3, where the blue region represents one equation like (6.23). Each row may be constructed such that the diagonal value of the matrix represents the coefficient to the center cell of the molecule. We know that the number of equations and the size of \( p \) is the same and equals the grid resolution \( N \). Thus, the matrix \( \nabla^2 \) is of size \( N \times N \) and very sparse, since only seven values in each row are non-zero. Storing \( \nabla^2 \) in memory requires \( N^2 \) entries, but only \( 7N \) is actually relevant. Therefore it should be kept implicit, either stored in a compact form, or implicit represented in the code. We use the latter approach, representing the calculation of one row in a kernel. The derivation of (6.20) into an equation showing the constant coefficients, follows the same pattern as we just showed for (6.21). Therefore we just show the
### 6.2 Solving Linear Systems

#### Figure 6.2: A seven point molecule used to calculate the Laplacian operator. The black molecules represent the center cell and its six neighbours involved in equation (6.23).

\[
\begin{bmatrix}
\nabla^2 \\
\end{bmatrix}
\begin{bmatrix}
p \\
\end{bmatrix}
= \begin{bmatrix}
\nabla \cdot u \\
\end{bmatrix}
\]

#### Figure 6.3: The full matrix system \(\nabla^2 p = \nabla \cdot u\). Only seven values in each row of the matrix are non-zero.

The same holds for the y- and z-components. What we have shown in (6.23) and (6.24) is that the pattern is very similar, only with different coefficients. This will be utilized to construct a solver that takes the known variables as arguments and solves the equation.
So far we have illustrated the structure of the linear system and derived the equations to be solved for the diffusive and pressure terms in the Navier-Stokes equations. Though, we have not presented an actual approach to solve the equations. A wide range of methods to solve linear equations exists, we recommend [16] for a comprehensive examination of linear and non-linear solver methods. Solving the equations exactly using a direct method, such as Gaussian elimination or LU decomposition is possible. Unfortunately, both approaches have a complexity of $O(N^3)$ and the inaccuracy caused by the numerical model is expected to dominate the accuracy gained from such direct methods. Therefore direct methods is rarely attractive for CFD solvers, instead we turn our attention to iterative methods. An iterative method starts with a guess to the solution and iteratively improves the guess until a given criterion is true or a maximum number of iterations is performed. The next section introduces the concept of iterative methods and present the approach we use to solve (6.20) and (6.21).

### 6.2.1 Iterative Methods

First we present the concept of iterative splitting methods and then we show how the Jacobi iterative method is applied to the Poisson equations described previously. In compact form we define a system of linear equations as:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (6.25)$$

An iterative method starts with an initial guess, and continuously improves the guess. After $n$ iterations, we have the guess $x^n$. When $x^n$ does not equal the exact solution $x$, there will be a difference between $b$ and $\mathbf{A}x^n$. This difference is called the residual $r^n$.

$$\mathbf{A}x^n = b - r^n \quad (6.26)$$

Subtracting this equation from the previous we get an expression for the difference between the guess and the actual solution, i.e. the error at iteration $n$.

$$\epsilon^n = x - x^n \quad (6.27)$$

Alternatively we could also express the residual from the error:

$$\mathbf{A}\epsilon^n = r^n \quad (6.28)$$

When $x^n$ approaches $x$ the error will reduce and so will the residual. An iteration scheme at iteration $n + 1$, improving the current guess $n$, could look like:

$$\mathbf{M}x^{n+1} = \mathbf{N}x^n + \beta \quad (6.29)$$
For such an iteration method to converge, the residual must decrease and at total convergence the guess will no longer change, hence $x^{n+1} = x^n = x$. Substituting this into (6.29) yields:

\[
\begin{align*}
Mx & = Nx + \beta \\
Mx - Nx & = \beta \\
(M - N)x & = \beta
\end{align*}
\]

Hence, if the linear system from (6.25) should converge using the iterative method, we must have:

\[
A = M - N \quad \text{and} \quad b = \beta
\]

This is the basic concept of an iterative splitting method, to split the matrix $A$ into two parts $M$ and $N$, and then continually calculate:

\[
x^{n+1} = M^{-1}(Nx^n + b)
\]

We call it relaxation, when calculating a better guess. Iteration methods differ in the way $M$ and $N$ are chosen. For a method to be efficient the matrices should be chosen with care, such that rapid convergence is achieved. Convergence can be analyzed, based on the eigenvalues of $M^{-1}N$. In fact, the number of iterations needed to achieve a result below a given tolerance can be predicted. However, we will not utilize such convergence analysis in this thesis due to time constraints.

The simplest iterative method is called the Jacobi method. Though the relative low convergence rate, it is often used on parallel architectures, because it fits them very well. Each thread of a Jacobi method is independent of all other threads, which is often good property on parallel systems. The Jacobi method constructs the matrix $M$ as:

\[
\begin{align*}
M_{i,i} & = A_{i,i} \\
M_{i,j} & = 0, \quad \text{for} \ i \neq j
\end{align*}
\]

I.e. $M$ is the diagonal of $A$. This leaves the definition of matrix $N$ to be:

\[
\begin{align*}
N_{i,i} & = 0 \\
N_{i,j} & = -A_{i,j}, \quad \text{for} \ i \neq j
\end{align*}
\]

Collecting the pieces gives a Jacobi iterative scheme for the two Poisson equations.

A single iteration using the scheme from (6.34) and the coefficients for the pressure term in (6.23) gives the following pressure update equation:

\[
p^{n+1}_{i,j,k} = \nabla \cdot \mathbf{u} - \frac{1}{\Delta x^2} (p^n_{i+1,j,k} + p^n_{i-1,j,k} - 2p^n_{i,j,k}) - \frac{1}{\Delta y^2} (p^n_{i,j+1,k} + p^n_{i,j-1,k} - 2p^n_{i,j,k}) - \frac{1}{\Delta z^2} (p^n_{i,j,k+1} + p^n_{i,j,k-1} - 2p^n_{i,j,k})
\]

\[
\left( \frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} + \frac{2}{\Delta z^2} \right)
\]

(6.39)
Likewise we can express the iteration update of the diffuse Poisson equation from (6.24) as:

\[
\tilde{u}_{n+1}^{i,j,k} = u^{i,j,k} + \nu \Delta t \left( \frac{\Delta x}{2} \left( \tilde{u}_{n+1}^{i+1,j,k} + \tilde{u}_{n+1}^{i-1,j,k} \right) + \frac{\Delta y}{2} \left( \tilde{u}_{n+1}^{i,j+1,k} + \tilde{u}_{n+1}^{i,j-1,k} \right) + \frac{\Delta z}{2} \left( \tilde{u}_{n+1}^{i,j,k+1} + \tilde{u}_{n+1}^{i,j,k-1} \right) \right)
\]

(6.40)

The Jacobi solver fits the GPU hardware well because each thread only needs to update one value, so no conflicts will occur. However, since the thread execution is parallel, a temporary copy of the vector field needs to be made, so that the solver reads from one and writes to the other.

### 6.2.2 Improving with Multigrid

The Jacobi method is simple and cheap, but it does not converge rapidly. A poor solution to the pressure field would cause the flow to be divergent and thus inaccurate. A simple solution could be to use more iterations or to use a stop criteria. Since we hope to achieve interactive simulation, using more iterations is not optimal. Using a stop criteria is neither a good idea, because then the time used by the linear solver is not limited and no time guarantees can be given. Besides, calculating a stop criteria might not even be trivial.

This suggests that we should either use another linear solver or improve the technique. Multigrid is a technique that applies to an existing iterative solver, and improves convergence. A multigrid technique complements methods like Jacobi well, because it helps remove low-frequency errors, whereas Jacobi itself is good for high-frequency errors. We will give a short introduction to the multigrid method here, for a detailed introduction and analysis we recommend [6, 9].

As mentioned, the Jacobi method converges rapidly for high-frequency errors, but not for low-frequency errors. [6] presents some very illustrative examples of the Jacobi method on high- and low-frequency problems. The Jacobi method described above uses finite differences based on only the nearest neighbour cells. Therefore, information can only travel a very limited amount in each iteration. Thus, information that needs to be propagated from one end to the other, requires as many iterations as the size of the linear system. The inspiration to multigrid methods has been heavily inspired by these facts. The overall idea is to use only few iterations and then continuously restrict the linear system to coarser and coarser grids. Once the coarsest grid is reached, the results are interpolated back onto the finer grids and relaxation is performed again. Interpolation is repeated until the original resolution is reached.

Two questions rise, how do we restrict to coarser grids and how do we interpolate back on finer grids? Let \( \Omega_h \) denote the finest grid, \( \Omega_{h/2} \) the second finest grid, with two-times less samples in each direction, and so on. A projection operator \( I \), taking
a vector $x$, from one grid to another is defined as:

$$
x_h = I_h^2 x_{h/2} \quad (6.41)
$$

$$
x_{h/2} = I_h^{h/2} x_h \quad (6.42)
$$

How $I$ is implemented may vary, we will only describe how it is used in this thesis, and refer to [6] for alternatives. Restriction is easy, we just copy the values that remains in the coarser grid. Averaging adjacent values are sometimes also used, but according to [6], results are not guaranteed to be better, so we prefer to keep it simple. Figure 6.4 illustrates restriction of a one dimensional discrete function, the same techniques holds for higher dimensions as well. Interpolation is a bit more difficult, because some of the values in the fine grid have no equivalents in the coarse grid. As the name suggests, interpolation should be performed for these grid points. In three dimensions, the following four scenarios occur for interpolation:

1. The new value can be copied directly from the coarse grid
2. The new value must be interpolated from two coarse grid points
3. The new value must be interpolated from four coarse grid points
4. The new value must be interpolated from eight coarse grid points

Figure 6.5 illustrates the four scenarios in the same order. In Section 6.2.1 we showed how the residual and the error of an iterative method relate:

$$
A_h \epsilon_h = b_h - A_h x_h^n \quad (6.43)
$$

This suggests, that when the first guess $x_h^n$ is obtained after $n$ iterations, we can calculate the residual, restrict it, and then solve for the error $\epsilon_{h/2}$ on the coarse grid, like this:

$$
r_{h/2} = I_h^{h/2} r_h \quad (6.44)
$$

$$
A_{h/2} \epsilon_{h/2} = r_{h/2} \quad (6.45)
$$
Chapter 6  6.2 Solving Linear Systems

Figure 6.5: Interpolation in three dimensions. Each column represents a different type of interpolation. The red and green nodes in the fine grid are interpolated the same way. The green nodes in the fine grids are examples of interpolations from the green nodes in the above coarse grids.

Once a better $\epsilon_{h2}$ is found, the fine grid results are corrected with the interpolated error:

$$\tilde{x}_h = x_h + I_{h/2}^h \epsilon_{h2}$$  \hspace{1cm} (6.46)

Where $\tilde{x}_h$ is the new adjusted guess. This list summarizes the multigrid method using one level of coarsening:

- Relax $A_h x_h = b_h$ with initial guess $x_h^0$.
- Compute the residual $r_h = b_h - A_h x_h$.
- Restrict the residual $r_{h/2} = I_h^{h/2} r_h$.
- Relax $A_{h/2} \epsilon_{h/2} = r_{h/2}$.
- Interpolate the error $\epsilon_h = I_{h/2}^h \epsilon_{h/2}$.
- Adjust the fine grid approximation $\tilde{x}_h = x_h + \epsilon_h$.
- Relax $A_h \tilde{x}_h = b_h$ with the updated $\tilde{x}_h$ as new start guess.

There is no need to stop after just one level, in fact multigrid often gives the best results if it continues until the coarsest grid is reached. The hole process might also be repeated two or more times with a different number of levels. Figure 6.6 illustrates three examples of what is called V-cycles. A V-cycle represents the coarsening strategi chosen for the multigrid method. There is no "correct" strategi to chose. In Section 8.2.1 we will test different V-cycles for both accuracy and efficiency, and compare it to the Jacobi method.

Solving the pressure Poisson equation accurate is more important than the diffusion equation. An inaccurate solution to the pressure equation will cause the velocity field to be divergent, creating unwanted flows and dissipation. Inaccurate diffusion will probably only cause the smoke to seem a bit thicker or thinner that it actually
6.3 Scalar Fields

Until now we have only been concerned with the velocity field update, but as mentioned a temperature and a mass field are also required to calculate the external forces. Hence, values from these fields must also be updated according to the current flow.

In Section 3.2 we derived the substantial derivative of a quantity \( \phi \) (also called the material derivative):

\[
\frac{d}{dt} \phi = \frac{\partial \phi}{\partial t} + (\mathbf{u} \cdot \nabla) \phi \tag{6.47}
\]

The first partial derivative is called the local acceleration and is just a measure of how fast the quantity is changing at a fixed point. The second part is the advective acceleration and is caused by the velocity field in which the quantity moves.
The material derivative of a temperature field $T$, is given by the heat diffusion equation:

$$\frac{dT}{dt} = \kappa \nabla^2 T$$

(6.48)

Where $\kappa$ is the thermal conductivity, a constant material property. Inserting $T$ and the material derivative into (6.47) we get a partial differential equation describing how temperature flows.

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T - (u \cdot \nabla) T$$

(6.49)

It makes good sense that a hot object spreads hot air in the area close to it, caused by diffusion, while if a wind blows by, it will also advect the hot air. Fortunately, this equation is similar to the diffusion and advection terms in the Navier-Stokes equations. Hence, this equation can be solved using the same techniques as presented earlier.

To represent the smoke inside the grid we need a scalar field, which will store the amount of smoke present in each cell. To be consistent with literature, such as [26, 34], we call it the density field. Notice however, that density as presented in Section 3.1.1 is constant in both time and space. Thus, the density field is not actually density, but just a percentage representing the amount of smoke present in each cell. The density field is passive in the sense that the material derivative is zero. Thus, the only update needed for the density field is caused by the advective acceleration.

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla) \rho$$

(6.50)

### 6.4 Boundary Conditions

So far we have not said much about boundaries. Boundaries are important and quite difficult to handle properly, because they cause discontinuities in the numerical models. If simulations took place in an outside (unbounded) environment without any objects, boundaries could be totally neglected, but obviously such simulations would be of limited use. Instead we bound the simulation inside boundaries, called the external boundary. We can then think of the flow as being inside a box. Obstacles can also be placed inside this box, creating internal boundaries.

Boundaries are represented using a grid of the same resolution as the simulation grid. Thus, a cell is either entirely occupied by an obstacle or entirely free for fluids to flow in. The outer boundaries are always present, and defined as:

$$c_{0,j,k}, \quad c_{w-1,j,k}, \quad c_{i,0,k}, \quad c_{i,h-1,k}, \quad c_{i,j,0}, \quad c_{i,j,d-1}, \quad \in \mathcal{G}$$

(6.51)

Where $w$ is the width, $h$ is the height, and $d$ is the depth of the grid. The actual location of the border between the boundary and the fluid is at the face between
the two adjacent cells. Figure 6.7 illustrates the boundary location in red. By controlling the values at the boundary cells, we can control the flow behavior near the borders. Fluid should never enter the boundary cells, i.e. the velocity normal at the borders must be zero. The tangential components can be used to determine the friction between the fluid and the boundary. If the tangential components are set to zero, it is called a no-slip condition, because the flow will then be totally still at the boundary. Tangential values other than zero can be used to control a frictional behavior. Such frictions are not actually physically correct, which is why we will only use no-slip conditions in this thesis.

The borders are located on the cell faces, but we can only set the values of the cell centers. Thus, the border is only implicit represented, see Figure 6.8. Interpolating the two explicit values in Figure 6.8 gives an expression for the value at the implicit
Chapter 6 6.4 Boundary Conditions

Using a no-slip boundary condition, we know that the velocity at $u_{\frac{1}{2},j,k}$ should be zero. Thus, we can get an expression for the boundary cell $u_{0,j,k}$, such that the implicit boundary condition is satisfied:

$$u_{0,j,k} = 2u_{\frac{1}{2},j,k} - u_{1,j,k}$$  \hspace{1cm} (6.52)

To generalize notation, the subscript "in" will be used for the values inside the boundary cells and "out" for values outside boundaries, i.e.

$$u_{in} = -u_{out}$$  \hspace{1cm} (6.54)

Satisfying boundary conditions in this way, setting the values directly in the velocity field, is called Dirichlet boundary conditions.

The pressure field should not cause any acceleration across boundaries. Hence, the pressure gradient over a border must be zero, i.e. $\nabla p = 0$. This is easily achieved by setting the boundary cells for the pressure field equal to the adjacent non boundary cell:

$$p_{in} = p_{out}$$  \hspace{1cm} (6.55)

Finally we must also control the boundaries for the temperature field. The motion of the temperature contains a diffusive term, which should not cause temperatures to accelerate into the boundaries. Again we therefore ensure that the derivative across the boundary is zero.

$$T_{in} = T_{out}$$  \hspace{1cm} (6.56)

Since the Navier-Stokes does not include any boundary information, solving each fractional step will break the boundary conditions. Thus, boundary conditions must be satisfied immediately after every alternation of the velocity, pressure or temperature field.

Finally the solver will handle ventilations attached to the outer border. The effect of a ventilator will be handled as a special boundary. Rearranging (6.52) and using the velocities for the ventilator we get:

$$u_{in} = 2u_{vent} - u_{out}$$  \hspace{1cm} (6.57)

The implicit velocities at the boundary will now equal the value specified by the ventilator velocity $u_{vent}$.
6.4 Boundary Issues

The method just presented raises some unsolved issues. First of all, a dilemma appears when obstacles are only one cell thick in one or more dimensions. The solution just presented, suggests that the value in the boundary cell should be computed only from the adjacent non-boundary cell. However, such a thin boundary will have adjacent non-boundary cells on both sides, see Figure 6.9. There is no obvious way to solve this issue. \[26\] presents an implicit boundary representation, where all readings from adjacent cells are first evaluated by the boundary constraints. Though the method did not slow down simulation speed, I fear that it will do in a CUDA environment, because it requires extra reading from global memory. Avoiding the problem, we simply require not to use one-cell thick obstacles. If thin obstacles are important, using a finer grid will solve the problem. A second issue, also illustrated in Figure 6.9 appears around corners. Once again there will be multiple adjacent non-boundary cells. Choosing an arbitrary adjacent value would create an incorrect boundary on at least one side of the corner. Instead we calculate the average of all adjacent non-boundary cells. The average value causes the corner to appear more smooth than sharp, this is however not a major problem, in fact it might be preferable in some situations. Calculating boundary values \( \phi \), for the temperature and pressure field now becomes an average as:

\[
\phi_{in} = \frac{1}{n} \sum_{i=0}^{n} \phi_{out}^i
\]

Where \( n \) is the total number of adjacent non-boundary cells. For the velocity field update, the summation is negated.

6.4.2 Boundaries and Multigrid

Introducing boundary conditions causes some considerably problems to the multigrid technique. First of all, boundary conditions must also be satisfied on the re-
stricted resolutions of the grid. Hence, the boundary field representing the boundary cells, must also be restricted. In three dimensions, restriction removes 7/8 of all cells, possibly causing obstacles to vanish. A thin wall would most likely vanish after only a few restrictions. Thus, the restricted linear system will be unaware of the wall, and the solution becomes incorrect.

Our solution is, to let the boundary cells be dominant, such that if any of the eight cells becoming one, is a boundary cell, then the restricted cell will also be a boundary cell. A two dimensional example is illustrated in Figure 6.10.

![Figure 6.10: Restriction of the boundary field. The outer border is always restricted to the new outer border. The red cells will be restricted to the next coarser level. The green cells will keep the boundary for the restriction, but only the dark green cell will hold the boundary for the dominant restriction. The initial obstacle vanished using normal restriction, whereas it remains for dominant restriction.](image)

shows that dominant restriction ensures that boundaries will never vanish. Restriction causes the boundary to change positions slightly. Therefore, solving the linear
system for the dominant case is not perfect either, but it is nevertheless much better compared to normal restriction.

Another issue with boundary conditions in connection with multigrid is also apparent in Figure 6.10. When an obstacle is restricted, it may lose one cell in size in the coarse grid, which we previously explained can not be handled by the boundary model. We will leave it to the user not to define obstacles that are too small, but as future work, this issue should be corrected. For example by using an implicit boundary representation as in [26] or at least letting the solver detect when obstacles get too thin and then stop restriction. The illustration also indicates that the boundaries rapidly make up most of the grid, so there is no need to restrict all the way to the coarsest levels. Stopping when the smallest dimension is around 16 should be sufficient. For a dimension of 128 cells, it would require that no obstacles are thinner than five cells.
Chapter 7

The Application

In the previous chapters we introduced CUDA and our approach to solve the Navier-Stokes equations numerically. In this chapter we present the application that completes the CFD solver.

The overall structure of the application is sketched in Figure 7.1. Each box is implemented as a separate class, such that it can easily be exchanged with alternatives. Above the red line is the initial input reading, discussed in Section 7.2. Once the configuration is build, the CFD solver starts execution of the fractional steps presented previously in Section 6.1. The outcome of the CFD solver is rendered to the screen and, if the user requests it, it will also be written to a text file. Visualization and data plotting is discussed in Section 7.4 and 7.5 respectively.

7.1 Setting up Grids and Blocks

In Section 5 we decided to use a collocated grid representation. Thus, all scalar and vector fields are of the same dimensions. CUDA delivers a set of allocation methods to allocate different types of memory. Since our solver solves for three dimensional flows, a three dimensional memory type is the obvious choice. Former GPGPU implementations based on shader techniques, such as [34], had to set up flat texture maps to imitate three dimensional textures. CUDA has made such abstractions easier.

The dimensions of the kernel grids and blocks should be created such that the kernel is executed by enough threads to manipulate the hole collocated grid. Some guidelines are proposed in [29], but they are all targeted for one or two dimensional problems. As mentioned earlier, grids cannot be three dimensional, so unfortunately these guidelines does not directly transfer to three dimensions. [4] presents an approach where threads are executed in planes and then the grid iterates through each level in the third dimension. The advantage is that more information can be stored in shared memory. However, multiple planes are required for each level,
Figure 7.1: The application layout. The simulation configuration is based on the FDS input file, parsed by the FDS reader. The CFD solver continuously simulate one time step and the visualizer renders the results. On user requests, output files are also written to the disk.

when mesh resolution is high, which makes indexing difficult. Furthermore, memory access requires intense jumping, which is not optimal according to the CUDA user guide. Whether this approach or others, could improve efficiency has not been tested. It would be an interesting topic in a future work to examine the characteristics of various setups.

We have designed an alternative approach that will make indexing easy. Each block will represent one ‘stick’ in the x-direction of the mesh \( G_{I,J,K} \). Hence, the blocks will be one-dimensional, with a size equal to \( I \). The grids will be two dimensional, representing the y- and z-direction of \( G_{I,J,K} \). Initializing the block and grid in the code looks as:

```c
1. dim3 block = dim3(I); // I threads
2. dim3 grid = dim3(K,J); // K*J blocks
```

Using the grid and block structure just presented, the xyz-coordinates can simply be defined as:

```c
1. #define XCOORD (threadIdx.x)
2. #define YCOORD (blockIdx.y)
3. #define ZCOORD (blockIdx.x)
```

These coordinates can be used to fetch values from a texture. To calculate the corresponding array index, the pitch value must be used. The following definition
returns the index based on the pitch value \( p \):

\[
\text{INDEX}(p) = (XCOORD + (YCOORD + ZCOORD \times \text{gridDim}.y) \times p)
\]

It should be clear from these definitions, that indexing uses a very sparse amount of calculations, and memory access would be sequential for each stick. We previously mentioned that shared memory is much faster than global memory. Shared memory are shared between threads in the same block, so whenever readings overlap for the same block, shared memory should be utilized. Most of our kernels are template functions, starting with the following lines:

1. \( \text{unsigned int } \text{idx} = \text{INDEX(pitch);} \) // Calculate index
2. \( \text{extern } \text{__shared__ } T \text{ neighbours[\()\];} \) // Declare shared memory
3. \( \text{neighbours}[XCOORD] = u[\text{idx}]; \) // Write from global to shared
4. \( \text{__syncthreads();} \) // Sync all threads

After these lines, the array \( \text{neighbours} \) should be used whenever accessing neighbour values in the x-direction, which is done every time finite difference is performed.

### 7.2 FDS Input File Format

FDS simulations use an input file to define all necessary properties of the simulation. To make the implementation as useful as possible in an engineering environment, we have chosen to base the configuration on the same input file format. Obviously FDS supports a wide range of parameters and functionalities that is not of any interest for our simulator. Therefore we will only support a subset of the FDS parameters.

The input file organizes each parameter in a namelist. Each namelist starts with an ampersand "&", followed by the namelist group name and ends with a slash "/". All relevant values of the current namelist group are listed inside these tags, separated with commas. Here is an example of a grid definition:

\[
&\text{MESH ID='grid', IJK=32,32,32, XB=0.00,1.00,0.00,1.00,0.00,1.00/}
\]

There is an comprehensive introduction to the file format and all of the namelist groups in [24], so we will not go further into depth with this. Our simulator supports the following namelist groups:

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>CHID</td>
</tr>
<tr>
<td>TIME</td>
<td>T_END</td>
</tr>
<tr>
<td>MESH</td>
<td>ID, IJK, XB</td>
</tr>
<tr>
<td>OBST</td>
<td>XB</td>
</tr>
<tr>
<td>VENT</td>
<td>SURF_ID, XB</td>
</tr>
<tr>
<td>SURF</td>
<td>ID, TMP_FRONT, VEL, VEL_T</td>
</tr>
</tbody>
</table>

There are some essential differences between our simulator and FDS. First of all, FDS is primarily a fire simulator, whereas our simulator is a smoke simulator.
Defining smoke using a surface, as fires are defined in FDS, turned out to give a very weak smoke source. Defining a smoke source as a volumetric box, like obstacles, gives the possibility to simulate a much more intense smoke. Furthermore the time step size is automatically adjusted in FDS, but not in our implementation. Therefore we extended the namelist groups with the following two additions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>DT</td>
<td>Time step</td>
</tr>
<tr>
<td>SMOKE</td>
<td>XB</td>
<td>Spatial location</td>
</tr>
<tr>
<td></td>
<td>TMP</td>
<td>Smoke temperature</td>
</tr>
<tr>
<td>SOURCE</td>
<td></td>
<td>Constant source or not</td>
</tr>
</tbody>
</table>

The FDS Reader class takes care of parsing the input FDS file into a configuration setup used by the CFD solver.

### 7.3 The CFD Solver

The CFD solver is the core of the application. It solves the Navier-Stokes equations using the fractional step approach presented in the Chapter 6. All updates of the velocity, pressure, temperature and density fields go through kernel calls from this class. Since the fractional method was described in details in Chapter 6, there is no need to present it here again. For implementation details we refer to the source code.

### 7.4 Visualization

As outlined in the project formulation, visualization is needed, but the quality is secondary. To save time, we sought a third party solution, and found a useful example in the online CUDA code samples called volumeRender. The example renders a three dimensional texture using ray casting. The final color is based on the number of non-zero ray penetrations.

After some modifications to the volumeRender, we created the smoke renderer. Our method is based on two steps, a pre-processing step and the actual rendering step. The pre-processing step fills a three dimensional texture with values from the density and obstacle fields. We would like to distinguish between smoke, obstacles and edge borders, so we fill the red, green and blue (RGB) channels with the respective values.

The render kernel casts multiple rays through the texture, one for each pixel to render on the screen. For each ray a number of samples are taken. Each sample fetches the RGB color from the texture, and accumulates the final color rendered on screen. Figure 7.2 illustrates an example using the original volumeRender from the CUDA samples and our modification.
7.4 Visualization

Figure 7.2: **Left:** The original three dimensional texture render from the CUDA code samples, called volumeRender. **Right:** A modification of the volumeRender where the RGB channels are used to represent smoke, obstacles and edges.

7.4.1 Limitations

Though the solution is very useful, it suffers from some drawbacks and limitations. First off all the volumeRender renders the textures as if they were cubic. When simulating fire in a room, the room is rarely totally cubic, which results in a skewed visualization.

As mentioned, the volumeRender casts rays for every pixel to be rendered on the screen. The time consumption therefore grows with $O(wh)$, where $w$ is the width and $h$ is the height of the render resolution. On one of the test machines, the difference between a 512x512 and a 1024x1024 resolution is actually noticeable. Shader and volumetric techniques are not implemented to give more realistic visual results. The current implementation renders all obstacles yellow, regardless of the view angle, which makes the obstacles look flat and reduces the impression from depth of field.

7.4.2 Slice Rendering

Not only the density field is interesting for an engineering purpose, so is the temperature and velocity fields. As an extension, we also implemented support for two dimensional rendering of the xy-planes of the vector and scalar fields. Stepping through the slices in the z-dimension i also supported.

The temperature field is rendered using a blue to red colormap, as seen in common engineering applications. The other scalar fields (density, pressure, and divergence) are all rendered in grey tones. The purpose of the latter two have merely been for debugging purpose. The velocity field is rendered in RGB colors, using the absolute values of the x-, y-, and z-component respectively.
7.5 Data Plotting

Visual representation gives a good intuitive representation of the smoke simulation, but for comparison and analysis purposes, the actual values of the data fields should be stored in data files. We implemented a data plotter, writing data into text files on the disk. Writing data to the disk requires the data to be copied from device to host, which is an expensive operation due to limited bandwidth. Plotting is therefore only performed on a request from the user. Once again we have reused an output file format much alike the FDS output format.

A plot consists of two initial lines specifying the name of the values and their respective units. All following lines represent the values from each grid cell. Plotting a slice of the velocity field could start out as:

\begin{verbatim}
X, Y, Z, U-VELOCITY, V-VELOCITY, W-VELOCITY
m, m, m, m/s, m/s, m/s
0.00000E+000, 0.00000E+000, 4.68750E-001, 3.57320E-001, 1.20711E+000, -7.47396E-005
3.12500E-002, 0.00000E+000, 4.68750E-001, 1.77736E-002, 1.71805E-002, 4.67206E-004
6.25000E-002, 0.00000E+000, 4.68750E-001, 6.76925E-002, 1.09579E-002, 4.11117E-004
9.37500E-002, 0.00000E+000, 4.68750E-001, 1.30852E-001, 8.24851E-003, 3.76676E-004
...
\end{verbatim}

Output files are stored in the same folder as the application and are uniquely named. To ease the process of analyzing the output files, we also created a small library of MatLab routines. Including reading data files into matrix representations and plotting both vector fields and streamline profiles. These MatLab files are also included on the enclosed DVD.
Chapter 8

Results

During the thesis we have discussed many test cases that would be interesting to perform and evaluate. However, it would not be appropriate to fill the whole thesis with such test cases, neither would there be the time. The following selection of test cases is a compromise of time constraints and what we believe will bring out the most essential and scientific innovative of this thesis.

All simulations are performed in three dimensions. However, both vector and scalar fields are better visualized and interpreted in two-dimensional plots and images. Therefore, we will generally only plot informations from the center slice of the z-direction, unless else is mentioned. Videos of selected test simulations can be found on the enclosed DVD under the test folder.

8.1 Fractional Tests

The first tests are basic confirmations that each fraction of the solver behaves intentionally. Each test examines a specific part of the solver and discusses the outcome with respect to the given hypothesis. The fractional tests include the following four:

• Exclusive Advection
• Exclusive Diffusion
• Pressure and Divergence
• External Forces and Boundaries

All these tests are performed with a 64x64x64 grid resolution and a time step of 0.01s. If any solver parts do not work properly, it will be clearly evident in the upcoming more complex case studies, so we will not spend much time on these basic tests, but merely confirm their behavior.
8.1 Fractional Tests

8.1.1 Exclusive Advection

We want to test the advection step with no other fractions active, i.e. there will be no diffusion, no pressure projection and no external forces. In the center of the grid we create a cube of upward pointing velocity vectors.

**Hypothesis:** Because of the Semi-Lagrangian advection scheme, velocities should decrease in strength from bottom up and never leave the initial area.

**Result:** Figure 8.1 illustrates the velocity field in RGB colors with 0.08s intervals. Semi-Lagrangian advection only updates values in the cells that already contain non-zero velocities. Consequently the velocities will never spread from their initial area. Also notice, that the zero-valued velocities in the bottom are slowly advected into the green area, i.e. advection alone introduces smoothing and dissipation. The result is as expected. The video simpleAdvection.avi shows the advection test.

![Velocity Field Example](image)

**Figure 8.1:** Semi-Lagrangian advection never advects the velocities outside the initial region and slowly smooths the velocity field. The RGB colors corresponds to the absolute values of the xyz vector components respectively, i.e. the green colors equal vertical velocities.
8.1.2 Exclusive Diffusion

This time we test for diffusion, the setup is the same as before, except we turn off advection and turn on diffusion.

**Hypothesis:** Velocities should slowly diffuse uniformly from their initial area, and decrease in sizes.

**Result:** Figure 8.2 illustrates the diffusive behavior of the velocity field with 6.0s intervals. Diffusion behaves as expected. The video simpleDiffusion.avi shows the diffusion test.

![Diffusion Behavior](image)

**Figure 8.2:** Diffusion creates a blurring effect. Viscosity determines the speed at which fluid diffuses, smoke diffuses rather slowly.

8.1.3 Pressure and Divergence

Once again we use the same setup, with upward pointing velocities in a centered cube. Pressure projection will be enabled, but no diffusion, no advection, and no external forces. The simulation is started in a divergent state, pressure should therefore ensure convergence towards a divergence free state.
**Hypothesis:** Pressure should build up, such that velocities are sucked from the bottom and up, creating swirls on each side of the middle. The divergence should converge to zero.

**Result:** Figure 8.3 illustrates the evolution in the first three time steps. Unfortunately the divergence is not totally zero, even after all three time steps. How fast it converges depends on the accuracy of the solution to the pressure Poisson equation. In Section 6.2.2 we investigate both the Jacobi and the multigrid method for convergence when solving the pressure equation.

Just as expected the pressure projection creates two circular movements around the middle, as depicted by the velocity profile in Figure 8.4.

The video pressureProjection.avi shows the pressure test.

---

**Figure 8.3:** The velocity field and the corresponding divergence field and pressure field. In the gray scale images, light gray equals positive divergence/pressure and dark gray equals negative divergence/pressure.

---

### 8.1.4 External Forces and Boundaries

In this setup, we unite all the fractions and test them together at once. We use the temperature and density field to apply external forces. In the middle of the grid we
insert a cubic cloud of hot smoke. The smoke temperature is set to 600°C and the ambient temperature is 0°C.

**Hypothesis:** The Buoyancy forces should overrule the gravity forces and result in upward velocities. Pressure projection should create circular movements as before, while advection raises the smoke and temperatures. As the temperatures diffuse and decrease, gravity forces should take over and make the smoke fall again. Boundary conditions should keep the smoke inside the grid at all time.

**Result:** The smoke cube rises in the well-known mushroom shape, due to the Buoyancy forces, as illustrated in Figure 8.5. As expected the temperatures diffuse rapidly until almost gone. Once the temperature equals the ambient temperature, the smoke falls down along the walls. Details from the whole simulation can be seen in the video allFractions.avi. The total amount of smoke seems to slowly dissipate, we expect this to be a consequence of the semi-Lagrangian part of the advection step. Advected values will never be larger than any of the previous values, i.e. the only possibility is that values decrease. The plot in Figure 8.6 illustrates how the total amount of smoke develops. When the flow has many fine vortices, we expect the density loss to be more evident, because the semi-Lagrangian step will then advect smaller values into the density area, see Figure 8.7 for an example. This could be the explanation to the break in the density curve after one second, when the smoke reaches the ceiling and creates new vortices. The fractional tests just presented all behaved as expected. We now turn to some more advanced and interesting test scenarios.
8.2 Case Studies

The following case studies reproduce some more complex scenarios, some of them will be comparable to results presented in previous literature. The purpose is to evaluate the correctness and reliability of our CFD solver. The first tests are performed to evaluate the improvement of the multigrid solver. According to our knowledge, no prior work has ever been presented, using multigrid in a CUDA based CFD solver. We will therefore have extra focus on the multigrid solver. The next test cases are all well studied in previous CFD papers. We have chosen these cases because they evaluate essential properties of the solver and because data

Figure 8.5: A cube of smoke at 600°C rises due to Buoyancy forces and drops again when the heat is gone. Left: Three dimensional smoke rendering. Middle: Density slice from the center of the z-dimension. Right: Temperature slice from the center of the z-dimension.
from other papers exists for comparison. Some of these studies was made in only two dimensions. To produce comparable results, we will therefore use free-slip boundary conditions on the walls in the spanwise direction (z-direction), such that the third dimension will hopefully be negligible. The last test will be a comparison between our solver and FDS. The case studies in topics will be:

- Multigrid vs Jacobi
- False Diffusion
- Shear-Driven Cavity Flow
- Flow Around a Mounted Cube in a Channel
8.2 Case Studies

8.2.1 Multigrid vs Jacobi

The purpose of this test case is to evaluate whether or not the multigrid approach, presented in Section 6.2.2, improves the solution to the pressure Poisson equation and thus the numerical and visual quality.

For the first test we setup an initial velocity field that is not divergence-free. The corresponding divergence field will be a mixture of low- and high-frequency values. The convergence videos on the DVD illustrates the initial divergence field. In Section 6.2.2 we stated that the multigrid approach removes low-frequency errors more rapidly than Jacobi. Thus, we expect to see improvements in this test. By the nature of an iterative solver, the residual of the linear system should converge towards zero. To compare the solvers, we calculate the norm of the residual for every iteration at the center slice in the z-direction. Only the pressure projection term will be active, so no other terms will have influence on the divergence field. In Figure 8.8 we have compared the Jacobi solver with four different V-cycles of the multigrid solver using a 128x128x128 grid resolution. From the figure we see that all multigrid cases converge much faster than the Jacobi solver. It is also indicated that four levels of restrictions and interpolations give the best results. Notice that the time between the samples on the coarse grids is smaller than between the samples at the fine levels. This is because the size of the linear system is also smaller on the coarse levels. Figure 8.8 indicates that it is preferable to use as many levels as the initial resolution allows.

Using the same setup, we also test for a different number of iterations at each level, illustrated in Figure 8.9. Once again the figure confirms that multigrid outperforms the Jacobi solver. Based on the convergence plots in Figure 8.8 and 8.9 we conclude that the best convergence for multigrid is achieved with the maximum number levels allowed by the initial grid and using two to four iterations.

For the last test using the same setup, we measure the effect on the divergence field after each time step. The divergence should converge towards zero, in order to uphold the continuity equation. After each time step the norm of the divergence field is calculated and plotted in Figure 8.10. We believe that the oscillating effect caused by the Jacobi solver is created due to pressure that is build up when solving the linear system. The pressure field from the previous time step is used as the initial guess, called warm starting. Once the continuity constraint is almost satisfied, the Jacobi method suffers from the spare amount of information that can only be shared by adjacent cells in each iteration, so the pressure can not be stopped immediately. On the DVD there are two videos illustrating the oscillating convergence of the Jacobi solver and the much faster multigrid convergence (JacobiConvergence.avi and MultigridConvergence.avi).

We have now showed that multigrid solves the Poisson pressure equation remarkably better and faster than the Jacobi solver. After about 25 iterations, the Jacobi
Figure 8.8: All multigrid V-cycle strategies beat the Jacobi solver considerably. The jumps in the multigrid samples are caused by restriction and interpolation to coarser and finer grids. A smaller linear system will have a smaller residual. The more restrictions the multigrid solver performs, the better the result is. The initial grid is 128x128x128, so the coarsest relaxations are performed on a 8x8x8 grid (for the red samples). Time is measured on the GPU only when the iterations kernels are executing.

solver only reduces the norm of the residual very little for each iteration. At that same time, multigrid has converged to a residual with a norm that is more than twice as good. But, will faster convergence and better accuracy be noticeable in an actual smoke simulation? The previous tests did assemble a rather designed setup, which we expected the multigrid solver to handle well. Furthermore, only the pressure term was active, so any numerical errors from the other terms was not included in the divergence field. Therefore, for the next test we setup a more complex scenario and observe the differences, both visually and numerically. The setup includes two ventilations, blowing air in from the bottom left side and top right side. In the middle we place an obstacle with a smoke source under it. The simulations can be seen in Multigrid.avi and Jacobi.avi. Figure 8.11 illustrates how the norm of the velocity divergence evolves through the simulation. Once again multigrid consistently produce the best results. In Figure 8.12 we illustrate how the Jacobi solver can not solve for low-frequency variations as well as the multigrid solver. Finally we have a visual comparison in Figure 8.13. The vortices created around
Figure 8.9: All the multigrid strategies converge better than the simple Jacobi solver. The number of iterations is not as important as the number of levels as Figure 8.8 indicates. After 30ms we see that two and four iterations are almost equal, but they both beat the six iterations.

the obstacle seems to be finer for the multigrid method, but the smoke also looks thinner. We expect some of the thin smoke to be caused by the advection step, which we previously showed suffered from dissipation near small vortices. But we expect the primary difference to be caused by the solutions to the linear system. As explained in [16], a Jacobi method ‘diffuses’ the initial guess until a steady state (the solution) is reached. Therefore, when our Jacobi solver does not converge as well as the multigrid solver, the solution will be in a more diffusive state. The consequence is that the smoke appears more diffusive and lacks of details. This is exactly what we observe in Figure 8.13.

To summarize the tests, we have showed not only that our multigrid solver converge significantly faster than the Jacobi solver, but also that the effect on the divergence field and visual quality is remarkable better. Thin smoke is more apparent in the multigrid simulations because of the smaller vortices. An explicit turbulent model controls how energy is taken out of the flow when large vortices decrease in sizes (energy loss). Hence, dissipation caused by decreasing vortices in our solution, has a similar effect as a turbulence model. This is called an implicit turbulence model, it is of course not as accurate as an explicit model would be.
Figure 8.10: Norm of the divergence field, measured after each time step. Multigrid converges rapidly and flattens out after one second. The Jacobi does not converge as fast plus it creates oscillating effects.

Figure 8.11: Norm of the divergence at the center slice in the z-direction. The multigrid method solves the pressure Poisson equation better than the Jacobi method all through the simulation.
Chapter 8

8.2 Case Studies

Figure 8.12: Both methods handle high-frequency errors equally bad/good, but only the Jacobi solver suffers from low-frequency errors. The blue circles indicates areas with negative divergence and the red indicates positive divergence. Multigrid does not have such large areas with varying values. The effect is very distinct in the JacobiDivergence.avi and MultigridDivergence.avi videos.

Figure 8.13: Visual comparison of the two methods. The difference is not overwhelming, but there are details to notice. The multigrid method seems to create some finer details/vortices around the obstacles. For the Jacobi solver we in general observe that the smoke is thicker and less detailed.

8.2.2 False Diffusion

False diffusion is numerical inaccuracy, introduced by the advection term, hence it often arise in advective dominated flows. Though diffusion is a natural part of the Navier-Stokes equation, diffusion caused by advection is not natural. If accuracy is not important for the fluid simulation (in games for instance), the diffusion term
is often skipped, and only false diffusion caused by advection appears. To test our advection scheme for false diffusion we turn off true diffusion and add cold and hot ventilators to the left and bottom sides of a cube. The ventilators will blow in the same diagonal direction and the opposite sides will be open, see Figure 8.14. If there was no false diffusion we would expect a sharp transition between the cold and hot fluid along the diagonal. But we know that the advection step performs interpolation of both the velocities and the temperatures, so instead we expect smoothing to appear and hence a soft transition. In [5] they show that advection of a quantity $q$, using the Semi-Lagrangian approach, actually equals solving the following equation:

$$\frac{\partial q}{\partial t} = - (u \cdot \nabla)q + u \cdot \Delta x \nabla^2 q$$

(8.1)

Which is advection plus an extra diffusion-like term, scaling with the grid cell size $\Delta x = (\Delta x, \Delta y, \Delta z)^T$. I.e. a fine grid should yield less false diffusion than coarser grids.

Figure 8.15 illustrates how the hot and cold temperatures mix due to false diffusion. As expected, false diffusion appears regardless of the resolutions, though better results are achieved for fine resolutions. A diagonal plot of the temperatures are plotted in Figure 8.16, which again conclude that false diffusion is most dominant in coarse resolutions. For future work, false diffusion could be minimized if the advection scheme was exchanged with a higher-order accurate scheme.
8.2 Case Studies

Figure 8.15: False diffusion using different grid resolutions.

Figure 8.16: Diagonal temperature profile. Finer grid resolutions give less false diffusion. The black line indicates the optimal solution.

8.2.3 Shear-Driven Cavity Flow

In this test a constant moving lid is simulated on top of a square cavity. The setup is illustrated in Figure 8.17. The cavity flow includes separation and reattachment of the flow, which often appear in practical situations of engineering interest. It is therefore an important property of the solver to correctly reproduce these scenarios. One primary vortex should appear in the middle together with two secondary vortices in the lower corners. The cavity flow is well studied in literature, see for
Table 8.1: $x$- and $y$-lengths of the two secondary vortices compared to [38]. As the resolution increases the results becomes closer to the reference results. All lengths are relative to $L$.

<table>
<thead>
<tr>
<th></th>
<th>32x32</th>
<th>64x64</th>
<th>128x128</th>
<th>256x256</th>
<th>Ghia et al. (129x129)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{x}^{1}$</td>
<td>0.12</td>
<td>0.16</td>
<td>0.19</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>$V_{y}^{1}$</td>
<td>0.10</td>
<td>0.12</td>
<td>0.14</td>
<td>0.16</td>
<td>0.18</td>
</tr>
<tr>
<td>$V_{x}^{2}$</td>
<td>0.32</td>
<td>0.29</td>
<td>0.30</td>
<td>0.31</td>
<td>0.30</td>
</tr>
<tr>
<td>$V_{y}^{2}$</td>
<td>0.38</td>
<td>0.36</td>
<td>0.36</td>
<td>0.35</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Example [38][38][38]. We will compare our results to the ones presented by Ghia et al. [38], which is considered to be a good standard of reference. To compare results we use a Reynolds number as:

$$\text{Re} = \frac{u_{lid}L}{\nu} = 1000$$  \hspace{1cm} (8.2)

Figure 8.17: A cubic cavity with a constant moving lid at velocity $u = (u_{lid}, 0, 0)^T$.

We test the cavity flow with four grid resolutions and expect the best results for the highest resolutions. The results in [38] are all obtained from two dimensional simulations, so we apply free-slip conditions to the walls in the third direction, which should make the third dimension irrelevant. Figure 8.18 illustrates streamlines for the cavity flow using a 128x128x128 grid. We obtain a flow that is very similar to the one presented in [38]. For comparison we measure the sizes of the secondary vortices $V^1$ and $V^2$, results are listed in Table 8.1. The results obtained in Table 8.1 and the velocity profiles measured at the center lines in Figure 8.19(a) and 8.19(b) are quite good compared to the references from [38]. Resolutions of 128x128 and 256x256 yields very close results, whereas the lower resolutions suffer from smoothing. As previously underlined, there is a strong connection between low resolutions and the numerical errors caused by finite difference approximations. Thus, the results are as expected.
8.2.4 Flow Around a Mounted Cube in a Channel

For this test we would like to push the solver to the limit, and beyond. For this test we use Reynolds numbers so high that the flow will be turbulent. Our solver does not include a model for handling turbulent transitions, such as advanced models, like Direct Numerical Simulations (DNS) or Large Eddy Simulations (LES), see [39] [10] [3] for details on these topics. Therefore we do not expect our solver to be consistent with the reference material. As stated in [39], even advanced solvers can not always produce good results for setups like this.

The setup consists of a mounted cube in the middle of an open flow channel, see Figure [8.19]. All relative dimensions are listed in the figure. Though the geometry is simple, the structure of the vortices will not be.

To reproduce a fully developed flow at the inlet section, the velocity profile is modified using a parabolic profile like:

\[ u_{x,0,j,k} = \frac{j(2D - j)k(7D - k)}{3.5D} \]  

The Reynolds number and the inlet velocities are related as:

\[ Re = \frac{u_{in}D}{\nu} = 40000 \]  

Where \( u_{in} \) is two-thirds of the maximum inlet velocity x-component, as suggested by Armaly et al. in [2].

This test case is again a well studied scenario and several results have been presented, see for example [3] [35] [39] [28]. One primary vortex should appear on the
Figure 8.19: Configuration of the mounted cube in an open channel with entering velocity profile. All lengths are relative to the size of $D$

backside of the cube and the re-attachment length is compared to the references listed in [3]. In [35] they show that the turbulent flow should be highly three-dimensional, creating a horizontal horseshoe going around the cube starting at the front of it.

The results we are able to produce are quite unsteady, this is not uncorrect since the flow is turbulent, but it makes it more difficult to compare results. A time-averaged velocity profile from the center xy-slice is illustrated in Figure 8.20. Several vortices appear in front of the cube, two of them were also reported by Yakhot et al. in [39]. Table 8.2 list the coordinates for the two vortex centers, compared to [39]. There is a very good agreement for the positions of the two centers, but Yakhot et al. use a Reynolds number of only 5610, so there is no guaranties that it is the same phenomenon we observe. However, it is a good explanation to why we observe more vortices than Yakhot et al. On the backside of the cube we do not achieve as good agreements to the reference material. Bennetsen summarizes
Figure 8.20: Time-averaged streamlines in the xy-plane over a mounted cube using a 256x128x256 grid. The flow was sampled 50 times with 0.04s intervals. The flow is three dimensional, which is indicated by the gathering streamlines behind the cube.

Table 8.2: Comparison of the x and y coordinates of the two vortex centers in front of the cube, using the measurements from [39] as reference.

<table>
<thead>
<tr>
<th>Method</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yakhot et al.</td>
<td>($-0.045, 0.012$)</td>
<td>($-0.075, 0.007$)</td>
</tr>
<tr>
<td>CUDA Solver</td>
<td>($-0.051, 0.019$)</td>
<td>($-0.073, 0.005$)</td>
</tr>
</tbody>
</table>

Table 8.3: Re-attachment lengths x_{re} comparison. The values are taken from [39] and will be referenced without further explanation. Our result is larger than all others, though it is not far from.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RANS with $k-\epsilon$ model</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANS with RNG $k-\epsilon$ model</td>
<td>2.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LES, Smagorinsky model</td>
<td>1.69</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LES, dynamic model</td>
<td>1.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solvek$\epsilon$ RANS</td>
<td></td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>LES Dynamic (no avg.)</td>
<td></td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>LES Dynamic 1-equation</td>
<td></td>
<td>1.67</td>
<td></td>
</tr>
<tr>
<td>Fractional Step</td>
<td></td>
<td>2.4</td>
<td></td>
</tr>
</tbody>
</table>

a list of measured re-attachment lengths for the primary vortex in [3]. The list is repeated in Table 8.3 along with our result. At the xz-plane measured just above the floor level, we get a streamline profile as illustrated in Figure 8.21. Though we experience some deviation compared to the reference profiles in Figure 8.22, there is an overall good equality between the characteristics of the flow. As in the reference profiles, two vortices are created on the back side of the cube. Taking into consideration, that our solver does not have any built-in turbulence model, we find the results satisfying. Furthermore, the present implementation does not have a model for handling turbulent wall boundaries, which will also affect the location of the re-attachment point behind the cube.
Figure 8.21: Streamlines from the bottom of the xz-plane around the mounted cube, using a 256x128x256 grid.

Figure 8.22: All four illustrations are reference streamlines by Nigro et al. [28], where a LES turbulent model was applied. The Reynolds number was 40000, similar to our test case. There is a good similarity to the streamline profiles we presented in Figure 8.20 and 8.21.
8.2.5 FDS vs CUDA Solver

The present solver has until now been shown to handle several test cases very well. For this final test, the solver will be evaluated in a more real-situation perspective, using FDS simulations as references. As mentioned, FDS solves a LES form of the Navier-Stokes equations, a highly advanced and complex solution approach. It is not expected from our solver to produce as accurate results, but if the overall flow and temperature distribution is alike, we consider it to be successful.

Two scenarios have been simulated, one simple setup and one that is more complex. The simple configuration only contains a fire/smoke source in the middle of an empty square room. FDS also simulates the fire, with a temperature that is hotter than the smoke and with a movement that creates a much more unsteady flow. Our current solver does not simulate fire, so there will be this unavoidable difference between the two.

In Figure 8.23 there is a comparison of the temperature fields after three seconds for the simple test. There is a fairly good agreement between the two, but as expected, there is much more distortion in the FDS simulation caused by the fire and the built-in turbulence model. After 10 seconds there is a smoke distribution as depicted in Figure 8.24. Again there is a clear effect of a missing turbulence model. The small scaled fluctuations that contributes to the diffusive behavior is not as visible in the CUDA solver. The more steady flow in the CUDA simulation has caused the smoke to be less spread, and are primarily present in the top region of the room. The FDS smoke has spread a little more in the room, but the similarity between the two are definitely recognizable.

The more complex setup includes a large flat obstacle in the middle of the room and two ventilators, both blowing ambient temperatures into the room. The ventilators will ensure that the flow is constantly disturbed, which should cause circulations. The temperature fields are compared in Figure 8.25 after 10 seconds. This time there is actually a better agreement between the two, because the ventilators create distortion so that the effect of the fire is less distinct. After 10 seconds there is a temperature around $250^\circ C$ in most of the area under the obstacle. Over the obstacles the temperature is less, around $150^\circ C$. In both simulations there is a clear indication of the ventilators blowing in ambient temperatures. The smoke distribution in Figure 8.26 is also very similar, there is a dense amount of smoke in the lower region, taking off as it rises above the obstacle. On the DVD there are several videos illustrating the two setups in different solutions.

In Table 8.4 and 8.5 the time consumption for the simple and the complex setups are listed. The simulations was executed by FDS on two different CPUs and by the CUDA solver on three different GPUs. This is where our CUDA solver really stands out. The biggest difference is for the 128x128x128 grid, where the CUDA solver is more than 300 times faster, using approximately four minutes, whereas FDS uses more than 21 hours! Though the computations are not directly comparable, it still gives a good hint of what speed ups that can be obtained when using the
GPU. Only the Tesla C1060 GPU with 4GB memory can handle a 256x256x256 resolution. It still does it in about half an hour, corresponding to approximately a half frame per second. For the 128x128x128 resolution, the Tesla GPU produces around five frames per second.

8.3 Summarizing the Results

From the fractional tests we showed that the individual parts of the solver performed as expected. The most critical defect is the continuous mass dissipation,
8.3 Summarizing the Results

Figure 8.25: **Left:** FDS Smokeview. **Right:** CUDA Solver. Temperature profiles, 10 seconds into the simulation. There is a good agreement between the two simulations. The temperature distribution is alike.

Figure 8.26: **Left:** FDS Smokeview. **Right:** CUDA Solver. Smoke distribution, 10 seconds into the simulation. There is a good agreement between the two simulations. The smoke distribution is alike.

but because of the first order approximations we should not expect better.

In the first case study the multigrid solver was evaluated and the results clearly underlined that both accuracy, visual quality and efficiency is gained compared to the Jacobi solver. The common conclusion from the last test cases are, that when the grid is sufficiently fine, the results are very accurate, compared to the reference material. Only under extreme conditions, where turbulence models are needed, the results were less comparable, but still within an expected range. The FDS comparison concluded that even though accuracy is not perfect, the time speed up is a powerful argument for future development of GPU based solutions. The next
### Chapter 8

#### 8.3 Summarizing the Results

**Table 8.4:** Performance comparison of the simple setup between FDS and the CUDA solver. Simulations was recorded in 10 seconds, with a time step set to 0.01s for all tests. For hardware details on the GPUs, we refer to the NVIDIA home page.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Simple</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resolution</strong></td>
<td>64x64x64</td>
<td>128x128x128</td>
<td>256x256x256</td>
</tr>
<tr>
<td>Intel @ 2.53GHz</td>
<td>5546s</td>
<td>75651s</td>
<td>-</td>
</tr>
<tr>
<td>Xeon @ 3.0GHz</td>
<td>5040s</td>
<td>84287s</td>
<td>-</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>39s</td>
<td>219s</td>
<td>2025s</td>
</tr>
<tr>
<td>GeForce 8800GTX</td>
<td>49s</td>
<td>332s</td>
<td>-</td>
</tr>
<tr>
<td>Quadro FX570M</td>
<td>445s</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 8.5:** Performance comparison of the complex setup between FDS and the CUDA solver. Simulations was recorded in 10 seconds, with a time step set to 0.01s for all tests.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Complex</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resolution</strong></td>
<td>64x64x64</td>
<td>128x128x128</td>
<td>256x256x256</td>
</tr>
<tr>
<td>Intel @ 2.53GHz</td>
<td>4328s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Xeon @ 3.0GHz</td>
<td>4233s</td>
<td>71663s</td>
<td>-</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>64s</td>
<td>251s</td>
<td>2128s</td>
</tr>
<tr>
<td>GeForce 8800GTX</td>
<td>72s</td>
<td>370s</td>
<td>-</td>
</tr>
<tr>
<td>Quadro 570M</td>
<td>520s</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

A step to make the present solver more comparable to FDS could be to include a turbulence model like LES. All in all we find the results very satisfying.
Chapter 9

Conclusion

In this thesis, we have combined the power of multiprocessor architectures on GPUs with a fractional step method for solving the Navier-Stokes equations. Based on CFD methods for real-time simulations, we have implemented a solver that compares well to referenced results of engineering standards. Accuracy has been improved with a multigrid solver and support for reading input files and writing data files was implemented to ease engineering work. The solution delivers functionalities for engineering purposes, such as smoke and temperature visualization and plotting.

Results presented indicate that our solver is comparable to approved fluid simulators. However, our solver is still in an early state and possible improvements have been pointed out and discussed. Dissipation caused by the first order schemes are significant when resolutions are low, whereas higher resolutions minimizes dissipation quite well. Thin boundaries limit the multigrid restrictions, due to the current boundary conditioning approach. Fortunately, both of these issues can be minimized when the resolution is high, which the solver has shown to perform well at. Though no explicit turbulence model was implemented, dissipation of the velocity field caused by the advection scheme actually works as an implicit model.

CUDA delivers a nice abstraction from previous shader programming techniques for GPGPU programs. However, in our opinion, it still seems to be in beta version at times. We ended up using much time on CUDA specific issues that was not straightforward and poorly documented.

During the thesis we have presented intermediate results to Rambøll and at the end we presented the results as they are given in this paper. Rambøll has shown great interest in the CUDA based solver and for the possibilities of future work. A key feature for Rambøll has been the possibility to get an interactive tool for CFD simulation. Though accuracy is not as high as current CPU based solvers, our solver would work as a great pre-processing simulator. Jens Chr. Bennetsen from Rambøll wrote an evaluation of the project, it can be found in Appendix A. He expresses that the solver can be of great value for Rambøll:
"The present CUDA solver can be applied for fast and rough check and thereby evaluate new building designs quickly. Furthermore, the speed of the present solver makes it possible to analyse several different locations of the fire sources thereby improving and making the building safer.”

We believe to have delivered a comprehensive presentation of a CUDA based smoke simulator and fulfilling the goals presented in the project description.

9.1 Future Work

Approaches for solving the Navier-Stokes equations are numerous. As outlined and discussed in this thesis, the fractional step method implemented has both advantages and disadvantages. For some of the possible improvements and ideas for future work we mention:

Staggered grid - Switching from collocated grid to a staggered grid could reduce numerical inaccuracy near boundaries.

Higher order differences - The first order differences used in both time and space could be exchanged with higher order schemes to improve the approximations.

Visualization - Though visualization is independent of the solver, good images help understand the flow. For example, it would be nice to render temperature slices directly in the 3D view and to have shading effects on the obstacles.

Double precision - Only few GPUs support double precision arithmetics, there have been no time to investigate the gain of using such double precision compared to single precision.

CUDA grids and blocks - Alternative setups for the CUDA grids and blocks would be interesting to investigate for and test for possible efficiency speedups.

Visibility - Another useful information for evaluating smoke risks is reduced visibility, caused by the smoke. Including this information in the data plots would be a nice feature.

Though, one must remember that most of these improvements would cause the solver to be less efficient, and in the end the interactive advantage could disappear. At the end of this thesis, our CUDA based CFD solver has already been used in several other projects at the Department of Computer Science. Two students are currently working on a solution for multiple GPUs. Another student is turning the smoke simulator into a water simulator and finally a PhD student is using our smoke data for realistic volumetric rendering effects.
Bibliography


Chapter 9

BIBLIOGRAPHY


[39] Alexander Yakhot, Tomer Anor, Heoing Liu, and Nikolay Nikitin. Direct numerical simulation of turbulent flow around a wall-mounted cube: spatio-
Appendix A

Rambøll Evaluation

The attached paper was written by Jens Chr. Bennetsen from Rambøll as a final evaluation of the project.
Evaluation of the master project: 
Smoke Simulation for Fire Engineering using CUDA

The implementation and use of performance-oriented fire codes and regulations within recent years has lead to an increased design freedom for constructing more spectacular and complex buildings, which not only pushes the structural design to the limit of what is possible, but also the structural response during a fire and the needs for improving the safety and reliability of buildings in such an event. 
The need for and use of fire dynamics simulations, also called Computational Fluid Dynamics simulations, and thermal/structural analyses that can describe the structural response during a fire scenario has increased over the last few years to enable the increasing complexity of new designs.

In general, the main purpose of the CFD simulations in this context is to predict the spread of smoke through the building space and the effectiveness of smoke venting measures to restrict that spread. The results are then assessed using some form of “tenability” criteria, typically based on a combination of visibility and temperature. All of this is used as a means of estimating the “available safe egress time”.

The presented approach to implement of a fast CUDA solver for smoke propagation is therefore very attractive. There has been several approach to make a fast smoke solver for computer graphics, but only a very few for actual smoke propagation for engineering purposes.

The present CUDA solver can be applied for fast and rough check and thereby evaluate new building designs quickly. Furthermore, the speed of the present solver makes it possible to analyse several different locations of the fire sources thereby improving and making the building safer.

The predictive capability of the present CUDA solver for smoke propagation is very sufficient. Also the sizes of the computational grid size of 256 x 256 x 256, make the solver very attractable. At Rambøll, the size of the computational fire models, range from 500.000 to 4.000.000 cells in most cases. The main limitation is the overall computational time required to complete a simulation which often range from several hours to several days, even when a parallel computation is applied.
The CUDA solver can however be extended with more advanced turbulence model, like an LES with Smagorinsky subgrid model and turbulent wall model to improve the agreement in very advance problems is required.

But the solver is not limited to internal smoke propagation. External problems like external fire, gas dispersion over terrain or urban scale, or at Oil and Gas installation is possible and feasible.

The present CUDA solver show an initial implementation, which can be extended to included more advanced physical models. But at the current state it is fully useful for our business.