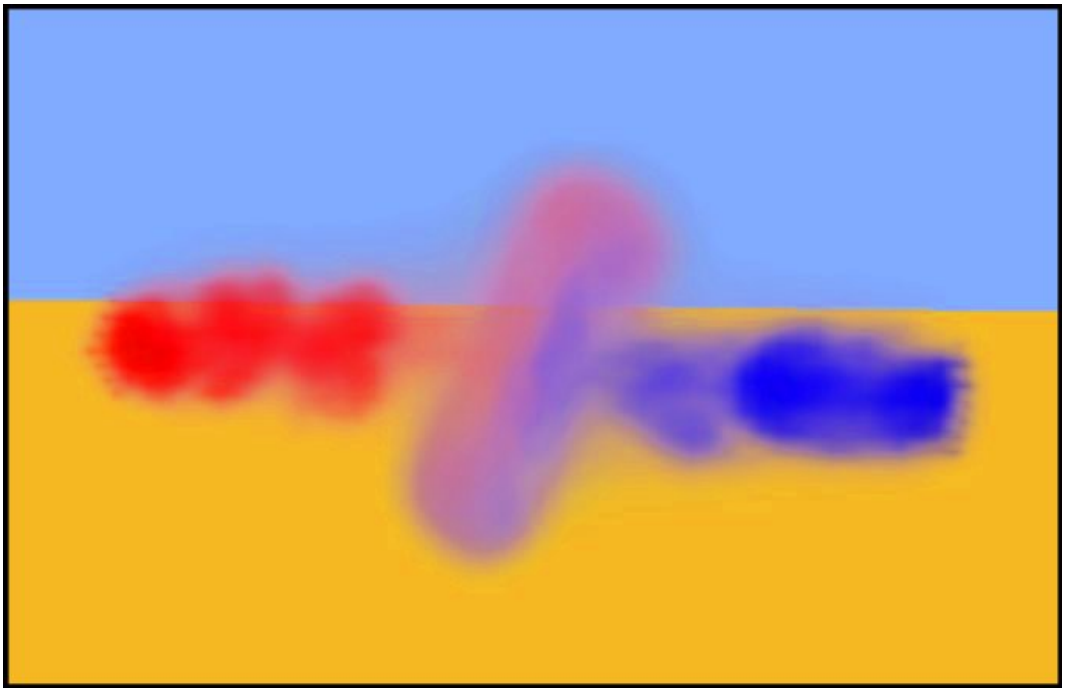


Real-Time Simulation of 3D Fluid Using Graphics Hardware

Masters Thesis in Computer Science
by
Marinus Rørbech

1st July 2004

Preface

This report is a Masters thesis in computer science at the Department of Computer Science, University of Copenhagen (DIKU), Denmark. It is the result of eight months work, and is estimated to a credit of 33 ECTS (corresponding to a little more than six months of full time studying).

The topic of this report is the simulation of fluid behavior and its implementation and execution using programmable graphics hardware. Since the field graphics hardware is very fast moving, the current standards have changed a lot, even in the rather short duration of this project. Thus, elements of this report may already be outdated at the time it is turned in. When reading, it should be kept in mind that this report only reflects a snapshot of the available methods and features, at the time of writing.

During my work with this project, others have published material on the same subject, some too late for me to take their work into account and appreciation. As a result of this, related or even dublicate work may exist, which have not been considered in the analysis. It is my conviction, however, that this report is still legitimate as an introductory text on the subject, and show legitimate results.

Acknowledgement

A lot of people have helped me in this work, and their efforts should not be forgotten. First of all I wish to thank my wife, Mette, who has stood up with me during the work of this thesis, and has been a great support on the occasional gray days (and nights). I also wish to thank my supervisor at DIKU, Kenny Erleben, and my co-supervisor at IO Interactive, Thomas Jakobsen, who have both been great sources of knowledge and inspiration. Special thanks also goes to Jesper Taxbøl, for help on the understanding of graphics hardware and fragment shaders.

Other people who deserve recognition include André Tischer Poulsen and Jens Egeblad for help with proofreading of this report, as well as Jacok Holck, Niels Ebsen Boldt, Jonas Meyer, and the rest of the guys from DIKU's Penthouse, for related discussions.

Marinus Rørbech

Contents

Chapter 1

Introduction

Within the past years, the computer gaming industry has exceeded the film industry in turnover, and most popular computer games are now sold in millions of copies all over the world. This growth in the market has increased the competition, and thus the need for generating new and unseen effects in games.

Hardware developers have extended graphics adapters with a wide range of features, to improve the overall visual quality in computer games. Most astonishing is the programmable *Graphic Processing Unit* (GPU) available on recent graphics adapters such as the NVIDIA GeForce FX series and the ATI Radeon 9000 series.

Along with the general increase in the power of standard home computers, the methods used for creating special effects have been improved, allowing more and more advanced effects. Today, special effects in computer games are often based on advanced mathematics and physics.

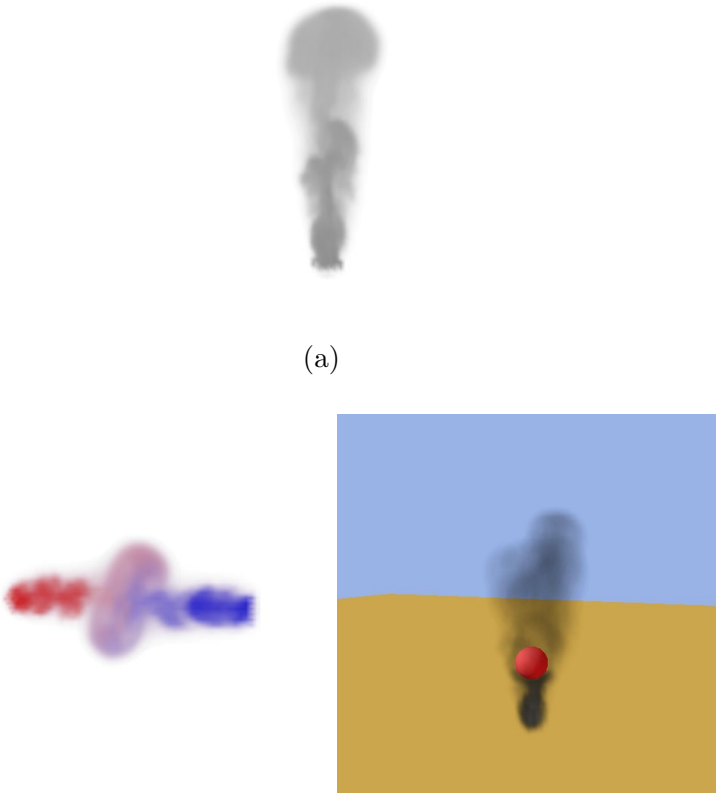
A category of computer game effects, which can still be improved a lot, is that of fluid effects, such as rising smoke, splashing water, fire, and explosions. Many ways of simulating fluid-like behavior have been tried, but even the most recent attempts on splashing water and rising smoke still require a great deal of imagination by the player.

1.1 This Work

With this report we wish to contribute to the research of making real-time simulated fluid effects possible in near-future computer games. We do this by describing a way to use the new features of modern graphics hardware, in order to make previous methods for fluid simulation run faster. We describe both the mathematical models of fluid motion, numerical methods for simulating these models with a computer, and a way to implement the numerical methods into *fragment shaders* run on a GPU.

Although many of the described elements can be used to simulate all sorts of fluid behavior, the duration and size of this project constrain us to focus on a single type of fluid effect. For this purpose we select the effect of smoke, in the way it can be observed from a bon fire.

As part of the project, the methods described in this report have been implemented into a demo application. Although the demo application is far from the quality and speed



(a)

(b) (c)

Figure 1.1: Images generated with the demo application. (a) Simple rising smoke. (b) Multi-colored smoke. (c) Interaction between the smoke and objects. The images are taken from animations simulated in real-time at approximately 20 frames per second.

required in a computer game, we prove with this application that real-time simulated fluid effects in computer games are just around the technological corner. Since this demo application is only a proof-of-concept, we will not give any introduction to its use. We refer readers to the movie clips and the `.bat` files on the accompanying CD-ROM, explained in section 1.4.

Figure 1.1 shows images created with the demo application.

1.2 Readers Prerequisites

Simulating fluid involves a great deal of mathematics. To explain all the mathematics used in this report, would be out of scope. Thus, readers are expected to have minimum mathematics skills, equivalent to the first year of an undergraduate study. The basics of discrete differentiation, which is used extensively throughout the report, can be refreshed in appendix A. For readers who are unfamiliar with iterative techniques for solving linear

systems, this, and its use in relation to this project, is introduced in appendix B.

Although some physics skills may be helpful, this thesis should be understandable for readers without any particular skills in physics. The basic physics of fluid dynamics is described in chapter 3.

To understand the use of fragment shaders in this project, some basic knowledge of graphics processing is required. This should be covered sufficiently in chapter 4. However, the reader is presumed to have a concept knowledge of basic operations, such as phong shading and texture mapping.

1.3 Structure of the Report

This report is divided into the following logical parts.

Requisites

- **Chapter 2:** First, we will introduce the notation used throughout the report.
- **Chapter 3:** To give the reader the necessary knowledge of fluid physics, this chapter will introduce the equation of fluid motion, and explain each term in details.
- **Chapter 4:** In this chapter, the fundamentals of 3D graphics will be described, and graphics hardware programming is introduced.
- **Chapter 5:** To give an idea of the problems involved in simulating fluid, and to benefit from already developed methods, a review on earlier work is given in this chapter.

Defining this Work

- **Chapter 6:** Having all prerequisites in place, this chapter will sum up the open issues of earlier methods, and based on these the goal of this work will be concretized.

Analysis

- **Chapter 7:** Here, the different ways to discretize the simulation volume will be described.
- **Chapter 8:** This chapter describes the steps involved in solving the equations of fluid motion.
- **Chapter 9:** Boundary conditions, which are necessary for solving fluid motion, will be described in this chapter, including the requirements for interaction between the fluid and objects.
- **Chapter 10:** To be able to present the simulations visually, this chapter describes simple ways to render the simulated fluid.
- **Chapter 11:** In this chapter, we will describe the implementation of the described methods into a demo application.

Round off

- **Chapter 12:** Based on the initial goals, we will set up some tests of the demo application, and evaluate the results.
- **Chapter 13:** Finally, we will round off this report with a conclusion and some ideas for future work.

Appendices

- **Appendix A:** The discrete computation of derivatives are used extensively throughout the report. This appendix shortly describe the concepts of first-order finite differencing.
- **Appendix B:** Solving linear systems is an important part of a stable fluid solver. This appendix introduces the concept of iterative solvers and concretize in the Jacobi method.

1.4 The accompanying CD-ROM

The original copy of this report is accompanied by a CD-ROM, which can found on the inside of the back cover. The CD-ROM contains the following material:

`/configs/<testname>.hws`	XML files defining the configurations of the tests.
`/movies/`	A few example movies in AVI format.
`/papers/`	A lot of the referenced papers in PDF format.
`/shaders/`	The fragment shaders implementing the fluid solver, described in section 11.3.
`/solver/`	The C and C++ files and headers implementing the demo application. These are ment only for browsing. Thus, no project, workspace, or makefile is supplied.
`/SetupGUI.exe`	The setup application used to create the scene files. This program is supplied without documentation
`/setupgui/main.cpp`	The C file implementing the setup application.
`/scenes/`	XML files defining the scenes used in the configurations of the test. These files define the size of the grids, and placement of the forces, source, etc.
`/SimpleSolver.exe`	The demo application. This is also provided without documentation. To run the tests use the `.bat` files. This application will only work on computers with an recent ATI graphics card.
`/test/<testname>.avi`	Animations from the tests, named by the respective test.
`/test/<testname>/`	JPEG images from the tests (one directory per test). Images are named by the respective test and frame number.
`/<testname>.bat`	`.bat`-files for executing the tests. These will only work on computers with a recent ATI graphics card. Double click files to run the respective test. The 'p' key starts and stops simulations.
`/thesis.pdf`	A digital copy of this report in PDF format.

Chapter 2

Notation

It has already been noted that mathematics play an important role in the simulation of fluid dynamics. In this chapter we introduce the notation used in this report, so the mathematics can be introduced and referenced in a consistent manner. Also, we introduce some mathematical concepts, which are used extensively throughout the report. Readers who are familiar with mathematical texts may find this chapter superfluous; others may want to use it merely as a reference.

2.1 Vectors and Vector Fields

To denote vectors we will use small caps and bold face, e.g. $\mathbf{v}$. Similarly, vector fields will be written in small caps and bold face, e.g. $\mathbf{v}(\mathbf{x})$. When nothing else is mentioned, vectors are assumed to be three-dimensional, i.e. $\mathbf{v} \in \mathbb{R}^3$, and vector fields are assumed to be functions $\mathbf{v}(\mathbf{x}) : \mathbb{R}^3 \to \mathbb{R}^3$.

Vectors and vector fields will sometimes be written componentwise. In this case we will use the notations

$$\mathbf{v} = (v^x, v^y, v^z)^T, \quad v^x, v^y, v^z \in \mathbb{R}, \tag{2.1}$$

$$\mathbf{v}(\mathbf{x}) = (v^x(\mathbf{x}), v^y(\mathbf{x}), v^y(\mathbf{x}))^T, \quad v^x(\mathbf{x}), v^y(\mathbf{x}), v^y(\mathbf{x}) \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^3. \tag{2.2}$$

Scalars and scalar fields are denoted by small caps in italic face, as exemplified by v^x and $v^x(\mathbf{x})$ in the above equations.

2.2 The Gradient Operator

The equations of fluid motion are highly dependent on the *gradient operator*, ∇ (nabla), defined by

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)^T. \tag{2.3}$$

For a scalar field $s(\mathbf{x})$ we define the *gradient* of the scalar field

$$\nabla s(\mathbf{x}) = \left(\frac{\partial s(\mathbf{x})}{\partial x}, \frac{\partial s(\mathbf{x})}{\partial y}, \frac{\partial s(\mathbf{x})}{\partial z} \right)^T, \qquad \mathbf{x} \in \mathbb{R}^3, \tag{2.4}$$

and for the vector field $\mathbf{v}(\mathbf{x})$ we define the *divergence* of the vector field

$$\nabla \cdot \mathbf{v}(\mathbf{x}) = \frac{\partial v^x(\mathbf{x})}{\partial x} + \frac{\partial v^y(\mathbf{x})}{\partial y} + \frac{\partial v^z(\mathbf{x})}{\partial z}, \qquad \mathbf{x} \in \mathbb{R}^3, \tag{2.5}$$

and the *curl* of the vector field

$$\nabla \times \mathbf{v}(\mathbf{x}) = \begin{pmatrix} \frac{\partial v^z(\mathbf{x})}{\partial y} - \frac{\partial v^y(\mathbf{x})}{\partial z} \\ \frac{\partial v^x(\mathbf{x})}{\partial z} - \frac{\partial v^z(\mathbf{x})}{\partial x} \\ \frac{\partial v^y(\mathbf{x})}{\partial x} - \frac{\partial v^x(\mathbf{x})}{\partial y} \end{pmatrix}, \qquad \mathbf{x} \in \mathbb{R}^3. \tag{2.6}$$

Throughout the literature on *Computational Fluid Dynamics* (CFD), and thus in this report, the gradient operator is used rather freely. For a vector field $\mathbf{v}(\mathbf{x})$ we use the notation

$$\nabla \cdot (\nabla \mathbf{v}(\mathbf{x})) = \begin{pmatrix} \frac{\partial^2 v^x(\mathbf{x})}{\partial x^2} + \frac{\partial^2 v^x(\mathbf{x})}{\partial y^2} + \frac{\partial^2 v^x(\mathbf{x})}{\partial z^2} \\ \frac{\partial^2 v^y(\mathbf{x})}{\partial x^2} + \frac{\partial^2 v^y(\mathbf{x})}{\partial y^2} + \frac{\partial^2 v^y(\mathbf{x})}{\partial z^2} \\ \frac{\partial^2 v^z(\mathbf{x})}{\partial x^2} + \frac{\partial^2 v^z(\mathbf{x})}{\partial y^2} + \frac{\partial^2 v^z(\mathbf{x})}{\partial z^2} \end{pmatrix}, \qquad \mathbf{x} \in \mathbb{R}^3, \tag{2.7}$$

and for vector fields $\mathbf{v}(\mathbf{x})$ and $\mathbf{w}(\mathbf{x})$, we use the notation

$$(\mathbf{v}(\mathbf{x}) \cdot \nabla)\, \mathbf{w}(\mathbf{x}) = \begin{pmatrix} v^x(\mathbf{x}) \frac{\partial w^x(\mathbf{x})}{\partial x} + v^y(\mathbf{x}) \frac{\partial w^x(\mathbf{x})}{\partial y} + v^z(\mathbf{x}) \frac{\partial w^x(\mathbf{x})}{\partial z} \\ v^x(\mathbf{x}) \frac{\partial w^y(\mathbf{x})}{\partial x} + v^y(\mathbf{x}) \frac{\partial w^y(\mathbf{x})}{\partial y} + v^z(\mathbf{x}) \frac{\partial w^y(\mathbf{x})}{\partial z} \\ v^x(\mathbf{x}) \frac{\partial w^z(\mathbf{x})}{\partial x} + v^y(\mathbf{x}) \frac{\partial w^z(\mathbf{x})}{\partial y} + v^z(\mathbf{x}) \frac{\partial w^z(\mathbf{x})}{\partial z} \end{pmatrix}, \qquad \mathbf{x} \in \mathbb{R}^3. \tag{2.8}$$

Finally, we use the notation

$$\nabla \cdot \nabla = \nabla^2. \tag{2.9}$$

2.3 Letters and Symbols

To ease the understanding of equations in the report we strive to use a consistent naming of physical coefficients and other properties that have special meanings. Table 2.1 lists some letters and symbols, which are used frequently throughout the report. Although scalars are otherwise denoted with small caps, in the case of temperature, T, we make an exception to avoid confusion with time, t, and to be consistent with other literature. Likewise, width, height, and depth are denoted with capitals to avoid confusion with the spatial descretization, h.

Physical coefficients

β	Thermal expansion of a fluid, describes the buoyancy per change in temperature
ρ	Density of a fluid, the mass of a unit cube of fluid
ν	Kinematic viscosity of a fluid, intuitively the "thickness" of the fluid

Solver related notation

t	Time
$\mathbf{p} \in \mathbb{R}^3$	A point in space
$\mathbf{u}$	A velocity field
$\mathbf{u}^t$	The velocity field at time t
$\mathbf{u}(\mathbf{p})$	The velocity at the point $\mathbf{p}$
$\mathcal{P}$	A path in a vector field
$\mathcal{P}(\mathbf{p}, \Delta t)$	The resulting point, when tracing point $\mathbf{p}$ along the path $\mathcal{P}$ for the time step Δt
D	Divergence of a velocity field, i.e., $D = \nabla \cdot \mathbf{u}$
p	A pressure field
$\mathbf{f}$	A force field
T	A temperature field
$\boldsymbol{\omega}$	A vorticity field
$\mathbf{n}$	A vorticity location field

Grid related notation

$\mathcal{G}$	A grid
$c \in \mathcal{G}$	A cell in the grid $\mathcal{G}$
h	The spatial discretization of a grid (i.e. the size of the cells in the grid)
W	The width of a grid (number of cells in x direction)
H	The height of a grid (number of cells in y direction)
D	The depth of a grid (number of cells in z direction)
$\mathcal{C}(c)$	The center of the cell c
$c_{i,j,k} \in \mathcal{G}$	The cell at position (i, j, k) in the grid $\mathcal{G}$, $0 \leq i < W,\ 0 \leq j < H,\ 0 \leq k < D,\ i, j, k \in \mathbb{Z}$
$\mathbf{u}_{i,j,k}$	The velocity in the center of cell $c_{i,j,k}$, i.e. $\mathbf{u}(\mathcal{C}(c_{i,j,k}))$
$u^x_{i-1/2,j,k}$	The x component of velocity on the left face of cell $c_{i,j,k} \in \mathcal{G}$, i.e. $\mathbf{u}(\mathcal{C}(c_{i,j,k}) - (h/2, h/2, h/2)^T)$

Table 2.1: Letter and symbol used frequently in the report.

Chapter 3

Defining Fluids

Since we want to simulate fluid, it is important that we understand what fluid really is. In this chapter, we will try to define fluid by describing how we percept fluid, and introduce the mathematical model of fluid motion.

To describe all physical properties of fluid would be out of the scope of this report. We will limit our introduction of physics to the extend required for understanding the report. For a thorough description of fluid physics, we refer the reader to physics textbooks, such as [30, 34].

3.1 Perception of Fluid

Fluid effects range from small scale effects, such as water poured into a glass and smoke rising from a cigarette, to large scale effects, such as weather systems and explosions. To give an idea of the wide range of fluid effects, we will give a list of examples, which we all will think of as fluid effects.

- **Evolution of Weather Systems:** Cloud formations, hurricanes, etc.
- **Global Wind:** Wind blowing in treetops, wind that makes fallen leafs "dance" in corners of buildings, etc.
- **Local Wind:** A blowing fan, warm air rising from a heater, etc.
- **Explosions:** From local explosions, such as from a grenade, to big explosions, creating blast waves.
- **Large Scale Liquids:** Rivers, oceans, waterfalls, etc.
- **Small Scale Liquids:** Coffee in a cup, water in a bath tub, etc.
- **Fire:** From candles to burning buildings.
- **Smoke:** For instance rising from a bonfire.
- **Mist:** Drifting fog.

Since the natures of many of these effects are very different, it is impractical to simulate them in the exact same way. To describe the simulation of all the mentioned effects would be an overwhelming job, and thus out of the scope of this report. On the other hand, many of these effects are related. Without wind, for instance, fog would not be drifting, and the difference between the effect of wind and the effect of an explosion blast wave is not very big – besides maybe in the power. As we shall see in chapter 5, the core of simulating many of the mentioned effects is the same. Thus, it seems reasonable to pick out a specific effect, and concentrate on the simulation of that. Chances are good that the same method will apply to other types of fluid effects.

> For the purpose of this project, we will focus on the simulation of smoke, as it is observed from a bon fire. This effect includes both circulation in a volume of air, and the distribution of some material – in this case the smoke – which covers many of above effects. Thus, when the word *fluid* is mentioned in this text, its primary meaning will be smoke. However, the reader should try not to think to narrowly of smoke, and keep in mind that other fluid effects might conform as well.

In chapter 13, we will briefly describe some details on how to extend the simulation of smoke, in order to cover some of the other effects mentioned.

3.2 The Navier-Stokes Equations

The *Navier-Stokes differential equations* [34] are a set of *partial differential equations* (PDEs), which describe the motion of viscous incompressible fluids[1]. This means that all properties of the fluid are described exclusively by its viscosity and density. Although air is not actually incompressible, most air effects observed in everyday life uphold this property. Thus, we will assume air to be incompressible in the context of this report.

The Navier-Stokes equations are based on the assumption that fluids can be described as a collection of particles. The terms describe the forces acting on a particle, derived by observing the behavoir in a unit cube around the particle. This is done under the assumption that the fluid inside this unit cube behave uniformly.

The Navier-Stokes equations were derived from Newton's second law of motion, which states that

$$\mathbf{f} = m\ \mathbf{a}, \tag{3.1}$$

where m is mass, $\mathbf{a}$ is acceleration, and $\mathbf{f}$ is force. They describe the changes in a velocity field, i.e. the acceleration of the fluid, as a sum of the forces acting on the fluid – including forces introduced by the fluids own movement.

In a compact vector notation, as used in many of the referred papers, the Navier-Stokes equations are presented as

$$\frac{\partial \mathbf{u}}{\partial t} = \frac{1}{\rho}\mathbf{f} - (\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} - \frac{1}{\rho}\nabla p, \tag{3.2}$$

$$\nabla\cdot\mathbf{u} = 0. \tag{3.3}$$

[1] Viscous incompressible fluids cover the most usual liquids, such as water.

In the following, we will explain the individual terms of these equations, to give an intuition of the contribution of each term to the motion of a fluid.

External Forces

A fluid can be affected by *external forces*. These are forces, such as gravity and wind, that are not generated by the fluid motion, but affect the fluid due to external conditions. The contribution of external forces is described by the term

$$\frac{1}{\rho}\mathbf{f} = \frac{1}{\rho}(f^x, f^y, f^z)^T \tag{3.4}$$

where the force field $\mathbf{f} = (f_x, f_y, f_z)^T$ is the sum of all external forces working on the fluid, and ρ is the *density* of the fluid, which describes the mass of a unit cube of fluid.

Advection

The force of the fluid motion working on the fluid it self is called *advection*. This can be thought of as the molecules in a fluid bouncing into each other. If one molecule bumps into another molecule, the other molecule is affected and will start moving. The contribution of advection is described by

$$-(\mathbf{u}\cdot\nabla)\mathbf{u} = -\begin{pmatrix} u^x\frac{\partial u^x}{\partial x} + u^y\frac{\partial u^x}{\partial y} + u^z\frac{\partial u^x}{\partial z} \\ u^x\frac{\partial u^y}{\partial x} + u^y\frac{\partial u^y}{\partial y} + u^z\frac{\partial u^y}{\partial z} \\ u^x\frac{\partial u^z}{\partial x} + u^y\frac{\partial u^z}{\partial y} + u^z\frac{\partial u^z}{\partial z} \end{pmatrix} \tag{3.5}$$

where ∇ is the gradient operator, and $\mathbf{u} = (u^x, u^y, u^z)^T$ is the velocity.

Diffusion

Diffusion occurs when part of the fluid passes by an obstacle, or another part of the fluid with a different velocity. The fluid is slowed down and vortices appear. The contribution of diffusion is described by the term

$$\nu\nabla^2\mathbf{u} = \nu\begin{pmatrix} \frac{\partial^2 u^x}{\partial x^2} + \frac{\partial^2 u^x}{\partial y^2} + \frac{\partial^2 u^x}{\partial z^2} \\ \frac{\partial^2 u^y}{\partial x^2} + \frac{\partial^2 u^y}{\partial y^2} + \frac{\partial^2 u^y}{\partial z^2} \\ \frac{\partial^2 u^z}{\partial x^2} + \frac{\partial^2 u^z}{\partial y^2} + \frac{\partial^2 u^z}{\partial z^2} \end{pmatrix} \tag{3.6}$$

where ν is the *kinematic viscosity* of the fluid, which describes the "thickness" of the fluid. Hence, diffusion is also refered to as the effect of viscosity.

Pressure

Fluid moving in and out of the observed unit cube causes the pressure to change. Differences in pressure between the unit cube and its surroundings affects the velocity as described given by

$$-\frac{1}{\rho}\nabla p = -\frac{1}{\rho}(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z})^T, \tag{3.7}$$

where ρ is still the density of the fluid and p is the pressure.

Incompressibility

To ensure that the volume of the fluid is kept constant, the net flow of the unit cube should be 0, indicating that the amounts of fluid entering and leaving the cube should be equal. This is described by the incompressibility constrant

$$\nabla \cdot \mathbf{u} = 0 \quad \Longleftrightarrow \quad \frac{\partial u^x}{\partial x} + \frac{\partial u^y}{\partial y} + \frac{\partial u^z}{\partial z} = 0. \tag{3.8}$$

By adding all these terms, the equations (3.2) and (3.8) can thus be written componentwise, without the use of the gradient operator, as

$$\frac{\partial u^x}{\partial t} = \frac{f^x}{\rho} - \Big(u^x \frac{\partial u^x}{\partial x} + u^y \frac{\partial u^x}{\partial y} + u^z \frac{\partial u^x}{\partial z}\Big) + \nu\Big(\frac{\partial^2 u^x}{\partial x^2} + \frac{\partial^2 u^x}{\partial y^2} + \frac{\partial^2 u^x}{\partial z^2}\Big) - \frac{1}{\rho}\frac{\partial p}{\partial x}, \tag{3.9}$$

$$\frac{\partial u^y}{\partial t} = \frac{f^y}{\rho} - \Big(u^x \frac{\partial u^y}{\partial x} + u^y \frac{\partial u^y}{\partial y} + u^z \frac{\partial u^y}{\partial z}\Big) + \nu\Big(\frac{\partial^2 u^y}{\partial x^2} + \frac{\partial^2 u^y}{\partial y^2} + \frac{\partial^2 u^y}{\partial z^2}\Big) - \frac{1}{\rho}\frac{\partial p}{\partial y}, \tag{3.10}$$

$$\frac{\partial u^z}{\partial t} = \frac{f^z}{\rho} - \Big(u^x \frac{\partial u^z}{\partial x} + u^y \frac{\partial u^z}{\partial y} + u^z \frac{\partial u^z}{\partial z}\Big) + \nu\Big(\frac{\partial^2 u^z}{\partial x^2} + \frac{\partial^2 u^z}{\partial y^2} + \frac{\partial^2 u^z}{\partial z^2}\Big) - \frac{1}{\rho}\frac{\partial p}{\partial z}, \tag{3.11}$$

$$\tag{3.12}$$

and

$$\frac{\partial u^x}{\partial x} + \frac{\partial u^y}{\partial y} + \frac{\partial u^z}{\partial z} = 0. \tag{3.13}$$

Chapter 4

Graphics Hardware

Along with the demand for higher quality in computer game graphics, as well as the demand for high performance in other graphics applications such as modeling and architectural design applications, special designed hardware for producing 3D computer graphics has become a more and more common part of a standard home computer. Some computers even have an advanced 3D-accelerated graphics adapter integrated directly on the main board.

Since graphics adapters are produced by different manufacturers, they work in different ways. In order for games and other applications to work with more than one specific device, several different *Application Programming Interfaces* (APIs) have been developed. Some of the most widely supported are OpenGL [29] and Microsoft DirectX [7].

As an extension to the standard capabilities offered by the adapters, most modern graphics adapters include an on-board processor called a *Graphics Processing Unit* (GPU), which allows developers to bypass the standard operations of the adapter, and perform their own specified operations.

In this chapter we will give an overview of the work performed by a standard graphics adapter and introduce some of the capabilitites of programmable GPUs.

4.1 Overview of the Render Pipeline

This section gives a short introduction to the process of rendering 3D graphics. Due to the author's own experience with 3D graphics, this description will be based on the OpenGL API as described in [1, 29]. For a detailed description of the subprocesses involved, we refer readers to standard graphics textbooks such as [11].

When rendering 3D graphics, the user application passes geometric primitives to the graphics adapter as streams of vertices. E.g., a triangle is passed to the graphics adapter as three vertices, defining the corners of the triangle. Each vertex can have a range of attributes attached to it, such as a normal vector, a color, and other attributes, which affect the processing of the vertex and the primitive it defines.

Upon receiving the vertex streams, the graphics adapter apply the appropriate geometric transformations of the vertex coordinates and normal vectors, to reflect the desired view of the 3D objects. A color is computed for each vertex, taking account for the at-

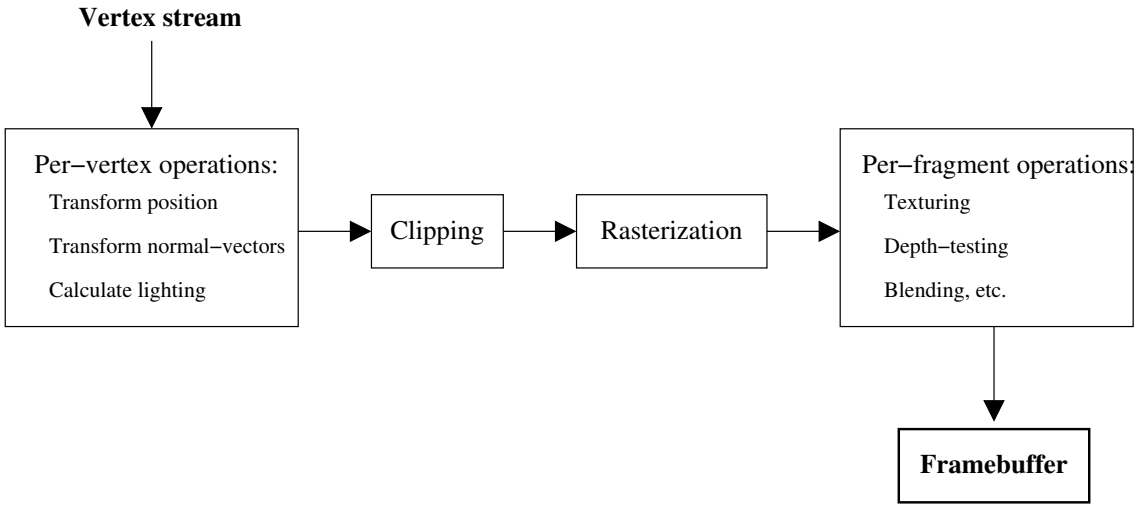


Figure 4.1: Schematic overview of the OpenGL render pipeline.

tributes of the vertices and the light in the scene. The primitives are clipped, so only the viewable parts remain, and the primitives are *rasterized*, producing streams of *fragments*, which are pixels with color values, texture coordinates, and depth values. Different operations, such as texture mapping, blending, and depth-testing, can then be performed for each fragment, before the fragments are finally drawn onto a frame buffer as pixels, thereby creating the rendered image. This process is known as a *render pipeline*, and is shown schematically in figure 4.1.

4.2 The GPU

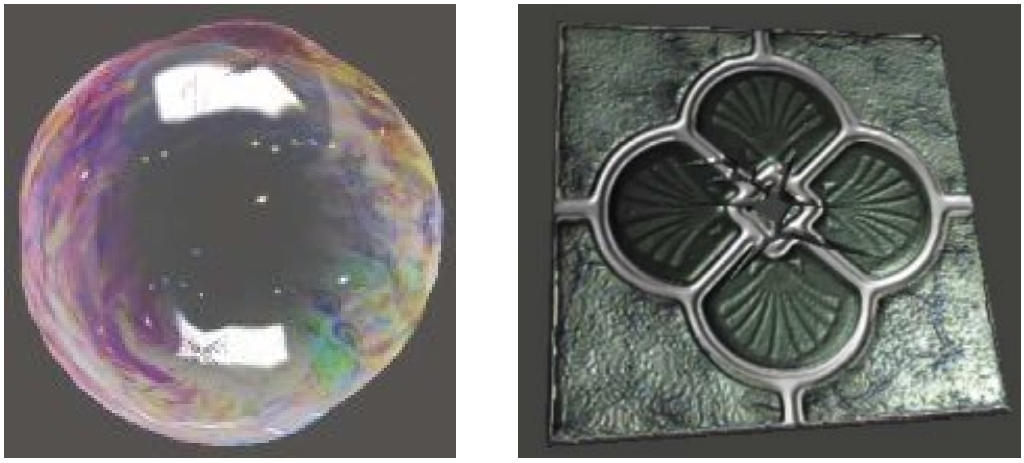
The render pipeline, as described above, requires a lot of state parameters to define the behavior of every step in the process. In the recent years, the increased demand for different features in the pipeline, such as rendering of fog and different kinds of advanced lighting calculations, has made this list of parameters grow.

To extend the possibilities of advanced rendering features without extending the list of feature parameters, and at the same time increase the general speed of rendering, modern graphics adapters, such as the Radeon series from ATI and the GeForce FX series from NVIDIA, have an on-board processor called a GPU.

4.2.1 Programmability

The GPU consists of two parts – a vertex processor and a fragment processor – which can be programmed separately. The programs are referred to as *vertex shader* and *fragment shaders*, respectively[1]. Figure 4.2 shows two examples of advanced rendering using vertex

[1]In the ealy phases of GPU programming, there have been some confusions on the naming of these programs. In OpenGL contexts, they have been refered to as vertex and fragment *programs*, while in DirectX contexts they are refered to as vertex and *pixel* shaders. To conform with the new specification



(a) (b)

Figure 4.2: Two examples of advanced rendering using a GPU: (a) A vertex shader is used to deform a sphere dynamically, creating a soap bubble-like effect. (b) Lighting and bump-mapping is implemented in a fragment shader, hereby achieving per-pixel bump-mapping and phong-shading. The images are taken from [3].

and fragment shaders.

Vertex shaders and fragment shaders are constructed using API specific languages. In OpenGL, these languages are supplied in the `GL_ARB_vertex_program` and `GL_ARB_fragment_program` extensions, defining two different assembly languages.

High-level languages for creating vertex shaders and fragment shaders are also becoming widely used. The most well known is Cg (C for graphics) [19], which is a C-like language designed by NVIDIA. The Microsoft DirectX 9.0 API supports a similar high-level language called High-Level Shading Language (HLSL) and in OpenGL the extension `GL_ARB_shading_language_100` gives support for the similar OpenGL Shading Language [18].

4.2.2 Differences in the Render Pipeline

Using vertex shaders and fragment shaders, changes the render pipeline dramatically. If vertex programming is enabled, the per-vertex operations mentioned in section 4.1 are not performed automatically. Thus, the required geometric transformation of vertices and normal-vectors, as well as calculation of colors and lighting, has to be performed by a user-supplied vertex shader. Similarly, if fragment programming is enabled, no per-fragment operations are performed automatically; any desired per-fragment operations have to be performed by a user-supplied fragment shader.

of the OpenGL Shading Language [18], we will refer to them as vertex and fragment shaders.

4.2.3 Shader Features and Limits

Since vertex shaders are run once per vertex issued to the graphics card and fragment shaders are run once per fragment of the rendered image, the programs are run millions – possibly billions – of times per second. Thus, fast execution of the programs is crucial. This has let to some limits, such as a maximum number of ALU instructions and texture read-instructions in the programs. These limits are hardware dependent and are generally loosened along with the development of new graphics adapters.

To decrease the overall running time of the programs, most new GPUs are designed with more than one pipeline for both the vertex processor and the fragment processor, so several vertices and several fragments can be processed in parallel. For this to be possible, some extra limits on the programs are necessary. The programs have to run the same commands in all pipelines, thus, no conditional looping or branching is allowed. Also, a fragment shader cannot write directly to the same memory as it reads from. This sometimes calls for more than one *render-pass* to perform a desired operation. As we will see in later chapters, this property also influences the range of iterative solvers possible to be implemented as fragment shaders.

The GPU is a vector-based processor, so most operations can be performed on four-dimensional vectors at the same speed as when performed on scalars.

4.2.4 Using the GPU for General Purpose Processing

Because of the speed of GPU operations, as well as the increased precision of the operations – new graphics adapters can operate on 32-bit IEEE single precision floating point numbers – people are starting to use the GPU for other things than just render-specific computations. Roughly speaking, if the state of some simulation can be represented with a texture map, the simulation can be updated by a fragment shader. However, special design of the states are often necessary, for the processing of the state with a fragment shader to be efficient.

By issuing drawing commands with proper texture coordinates, fragment shaders can be executed on specific parts of the data texture, and the result will be rendered to a frame buffer. The contents of the frame buffer can then be copied back into the texture. Some graphics adapters even offer the feature to render directly to a target texture.

Numerous examples of general purpose vertex and fragment shaders, can be found in [2, 15, 26].

Chapter 5

Earlier Works

Many different approaches to the simulation of fluid on computers have been suggested, ranging from simple methods, such as ad-hoc particle systems, to highly advanced, physically based methods. In this section some of the methods relevant for this project will be described, in order to give the reader an overview of the subproblems involved. Methods that are used in the analysis of this report will be explained in detail later, in their relevant contexts.

Some of the methods described in this chapter have not been explained in scientific documents, but are merely descriped based on our own experience from playing a lot of computer games, as well as our general interest in computer game efffects.

Although this thesis is about *real-time* simulation, a lot of the following methods are not actually real-time simulation methods. However, since these methods are based on combinations of physical properties of fluid and advanced numerical analysis, they produce much more realistic results than the real-time methods. Hence, to come up with a new, better way to simulate fluid in real-time, these methods are considered highly relevant.

Since many of the methods described in this section are similar, it would be tedious to explain them all in detail. To keep the number of pages down, and to save the reader from duplicate reading, only the important differences will be explained.

5.1 Ad-hoc Methods

The first uses of fluid effects in computer games were actually animated, rather than simulated. Animators would create small animations of diffusing smoke, which could be looped in order to give the impression that a torch produced smoke, which then diffused away. Although these animations are obviously non-dynamic, similar methods can been used to give impression of dust rising from the ground when a game character is running, thus, making it somewhat dynamic.

These animations have later been replaced by small *particle systems*, allowing for use in the upcoming 3D computer games. The effects no longer have to be animated, but can be generated dynamically. The particles in such a particle system would be moved upwards to give impression of buoyancy, and in some applications even moved randomly sideways to give impression of turbulence in the surrounding air. When displaying the

particle system, each particle can be drawn using a *bill board* – a plane polygon (often a quadrilateral), which is always facing the viewer. A texture map can be applied to the bill board, to make it look like a small cloud of dust or smoke, and to add the impression of diffusion, the bill board can be scaled in size – maybe even blended with the background – according to the particle position and lifetime.

Although very simple, this method is actually quite efficient, and many methods used in current computer games are based on this. Due to its simplicity the method is very fast, and it can easily be adapted to produce better visual quality, by e.g. adding more particles or by applying more advanced, maybe even physics-related, forces to the particles. However, the quality of this kind of simulation is still very limited since the number of particles cannot be increased infinitely. Furthermore, interaction between the fluid and the environment is not very efficient, since it requires detection of collisions between objects in the scene and the particles. Hence, methods like this are often restricted to small clouds of fluid, such as exhaustion from cars and smoke from torches, and cannot be used convincingly for fog and other large-scale fluid effects.

5.2 CFD-Based Methods

Along with the general increase in computational power over the last decade, the field of *Computational Fluid Dynamics* (CFD), and its use in computer graphics, has been more and more explored. This has lead to a number of methods based on advanced physical relations, such as the Navier-Stokes equations described in section 3.2. Although most of such methods do not run in real-time, the results achieved are very visually pleasing, and thus also desirable in computer games.

5.2.1 Foster and Metaxas – 1996

In [13], Foster and Metaxas describe one of the first methods for simulating the full 3D Navier-Stokes equations for computer graphics purposes. The simulation space is divided into equally sized cells, each holding the pressure in the center of the cell and the velocities on the faces of the cell, thereby representing discrete velocity and pressure fields. The effects of advection and diffusion are added to the velocity field by using finite differencing, resulting in a non-mass-conserving velocity field. Mass-conservation is upheld by adjusting velocities using an *Successive Over Relaxation* (SOR) method [9], until the net flow in each cell is lower than some threshold.

In order for the liquid to behave properly, boundary conditions are upheld during the update of the velocity field. The most advanced of these boundary conditions are applied to make the free surface of the liquid behave correctly – these are not relevant in the scope of this project. Other boundary conditions are applied in order for the fluid to interact correctly with objects constraining the fluid, e.g., walls keeping the fluid from entering specific parts of the simulation area.

Unfortunately, since this solver is based on explicit methods, it is only stable for small time steps. In fact, the time step must uphold the *Courant-Friedrichs-Levy* (CFL)

condition

$$1 > \max(u^x \frac{\partial t}{\partial x}, u^y \frac{\partial t}{\partial y}, u^z \frac{\partial t}{\partial z}), \quad \forall u^x, u^y, u^z \tag{5.1}$$

where u^x, u^y, and u^z are the velocity components in x, y, and z directions, respectively.

5.2.2 Foster and Metaxas – 1997

In [14], the method from [13] is extended to include the forces of thermal buoyancy. A temperature is represented in the center of each grid cell, defining a discrete temperature field in the same way as with pressure. The temperature field is advected and diffused using the same method as with velocity. When updating the velocity field, the temperatures are used for calculating external forces, given by

$$\mathbf{F}_{bv} = -\beta \mathbf{g}_y (T_0 - T_k), \tag{5.2}$$

where β is the coefficient of thermal expansion, $\mathbf{g}_y$ is the gravity vector, T_0 is the initial reference temperature, and T_k is the simulated temperature.

5.2.3 Stam – 1999

In [31], Jos Stam presented a method for solving the Navier-Stokes equations, where the explicit methods from [13] are replaced, in order to make the solver unconditionally stable. Thus, the different contributions of the Navier-Stokes equations are added in the following steps.

First, external forces are added simply by adding the force field to the velocity field. Second, the effect of advection is added by using a *semi-Lagrangian* method. With this method, grid points are advected backwards (as opposed to forward in Lagrangian methods) to find the new velocities values. Third, the effect of diffusion is added by solving the diffusion term using an implicit method, yielding

$$\tilde{\mathbf{u}} - \frac{\partial \tilde{\mathbf{u}}}{\partial t} = \mathbf{u}, \tag{5.3}$$

where $\mathbf{u}$ is the current velocity, $\tilde{\mathbf{u}}$ is the yet unknown, updated velocity, and $\frac{\partial \tilde{\mathbf{u}}}{\partial t}$ is given by the diffusion term of the Navier-Stokes equations, equation (3.6).

As in [13], this results in a velocity field that is not mass-conserving. But instead of using an SOR method, a mathematical result called *Helmholtz-Hodge decomposition* is used. This states that any vector field can be decomposed into a scalar gradient field and a mass-conserving vector field, yielding

$$\tilde{\mathbf{u}} = \mathbf{u} + \nabla q, \quad \nabla \cdot \mathbf{u} = 0, \tag{5.4}$$

$$\Rightarrow \nabla \cdot \tilde{\mathbf{u}} = \nabla^2 q. \tag{5.5}$$

Equation (5.5) is recognized as a *Poisson equation* for the unknown scalar field q. When spatially discretized this equation yields a sparse linear system, which can be solved in a number of ways [9, 6]. In [31] a *multigrid* method is used [6]. When solved, the pressure field is used to calculate the mass-conserving velocity field according to (5.4).

5.2.4 Fedkiw et al. – 2001

In [10], Fedkiw et al. presents a method for simulating smoke, based on the method from [31]. Under the assumption that the effects of viscosity are negligible in gases, when simulated on a coarse grid, the diffusion term of the Navier-Stokes equations are left out, leaving the incompressible Euler equations

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \mathbf{f}, \tag{5.6}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{5.7}$$

where $\mathbf{u}$ is the velocity, p is the pressure, and $\mathbf{f}$ is the external forces.

To make up for the numerical dissipation, introduced by the coarse grid representation, *vorticity confinement* [33] is used to add the small scale detail that has been damped out by numerical dissipation back into the velocity field.

5.2.5 Stam – 2003

In [32], Stam presents a simple version of his solver from [31]. This method is in fact so simple that the entire source code (approximately 100 lines of C) is presented in the article. Because of the simplicity and efficiency of this method, it can be implemented to run very fast. Thus, actual physically based simulation in real-time seems within reach.

5.3 Methods implemented on GPUs

Because of the complexity of the equations, as well as the numerical methods mentioned above, there is still a lot of work to do in the field of real-time fluid simulation. Lately, some attention has been given to the new generation of graphics adapters due to their architectural design for fast processing and the programmability of the GPU.

In [20, 4], examples of fluid simulation on GPUs are given by implementing matrix solvers as fragment shaders and representing matrices with texture maps. On the basis of the results achieved in these articles, simulation on GPUs seem like a good alternative path towards faster fluid simulation.

5.3.1 Harris et al. – 2003

In [17], Harris et al. describe a method for simulating clouds at interactive frame rates. The clouds are animated by simulating the fluid motion, taking both thermodynamics and water phase transitions into account. The fluid simulation is based on the methods presented in [10, 31].

To achieve interactive simulation rates, the simulation is implemented in fragment shaders running entirely on a GPU, thus taking advantage of the parallel capabilities, as well as the generally fast processing of data. To do this efficiently, simulation grids are represented as *flat 3D textures*, where each slice in the 3D grid is represented as a rectangular area on a 2D texture, see figure 5.1. In this way the entire 3D grid can be processed by the GPU in one render pass, simply by drawing one quadrilateral for each slice in the grid.

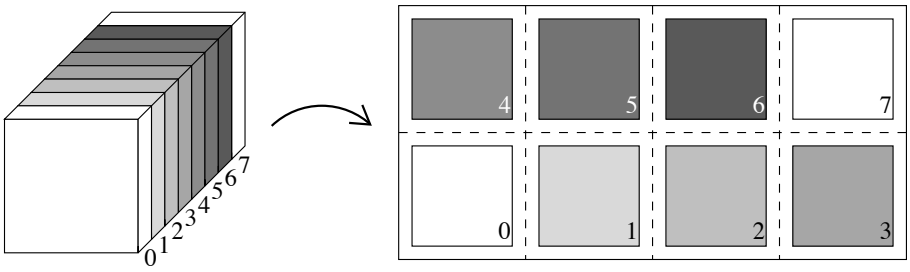


Figure 5.1: The slices of a 3D texture is laid out as tiles on a 2D texture. This figure is copied from [17].

Due to the inability of fragment shaders to read and write the same memory in a single render pass, methods such as *Gauss-Seidel* used in [32] and SOR used in [13] are not applicable. To address this, Harris et al. use a variation of the *Jacobi* method [9] (see appendix B) called *Red-Black Gauss-Seidel* [6], where grid cells are split into two sets (red and black) in such a way that the red cells can be updated using only black cells and vice versa. Furthermore, to reduce the number of texture lookups, pressure values are packed so that each *pixel* represents four pressure values.

The solver in [17] only runs at very low frame rates (about 1-3 frames per second). By taking advantage of the slow evolution of clouds, and because the semi-Lagrangiang advection scheme is stable for large time steps, they can however still use the solver in a real-time application, simply by only updating the simulation once per second. This is refered to as *time budgeting*.

5.4 Summary

Ad-hoc methods for simulating fluid effects in real-time do exist. Their use is, however, limited because of their simplicity.

More advanced methods, based on the physics of fluids, achieve much more realistic effects, and can be used for generating a wider range of effects. Unfortunately, they do not run in real-time.

Promising results have been made by taking advantage of new features of graphics hardware. However, real-time simulated fluid effects are not yet obainable in computer game engines.

With this overview in mind, we are ready to define the intensions of this project.

Chapter 6

Motivation

The method described by Stam in [32] is simple, stable, and efficient. However, when implemented to run on a CPU, the method does not run fast enough for real-time simulation of 3D effects, even on the most recent CPUs.

The method described by Harris et al. in [17] does run in real-time, but only due to the time budgeting and the slowly evolving property of clouds; the actual fluid simulation it self only runs at barely interactive rates. However, this model includes a lot of cloud-specific simulation, which can be left out to improve simulation speed.

6.1 Narrowing Down the Problem

The primary goal of this project is to benefit from the methods described in [17, 32], and implement a simple, stable, fluid solver, running on a GPU. Our hope is that fluid simulation, by this, can be made to run in real-time. The following list concretize this goal:

- **Simulation Method:** To generate realistically looking fluid animations, we will base our simulation on the Navier-Stokes equations, described in chapter 3. To do this, we will benefit from the earlier methods, described in chapter 5 – especially the methods described in [17] and [32] seem relevant.

- **Speed:** It is a key feature that the simulation runs at real-time rates. This will be prioritized higher than for instance the precision of the simulation. To achieve this, we will implement the solver in fragment shaders, for execution on a GPU of a modern graphics card.

- **Type of Fluid:** Many different types of fluid effects can be simulated using the Navier-Stokes equations. In order to limit the amount of work to fit the intended size of this project, we will concentrate on a single type of fluid, rather than on fluids in general. As defined in chapter 3, we will concentrate on the simulation of smoke. Since speed has a very high priority, we will not demand the produced animations to be realistic enough to delude anyone. However, efforts should still be put into making the animations convincing. I.e., people should be able to tell that it is smoke.

- **Application:** An obvious target for real-time fluid simulation, is the application into a computer game engine. Although this maybe too high a goal, given the current available hardware, this should be kept in mind throughout the analysis. Since implementation into a computer game introduces some very strict limits, we will not design our implementation to fit a computer game design. However, implementation of the described methods should be applicable to a computer game design.
- **Interaction with Objects:** An important feature, for use in computer games, is the ability of the simulated fluid to interact with a predefined scene. Interaction with moving objects is also highly desirable. However, since the complexity of getting the simulation up and running is uncertain, fluid/object interaction will only be considered if time allows it.
- **Visualization:** Although visualization is a key issue, when using the fluid simulation in computer games, realistic visualization is not a primary goal of this project; focus should be on the simulation. Visualization of the simulations should, however, be implemented to some minimum extend, to make evaluation of the simulations possible. If time allows it, visualization possibilities should be explored more deeply.
- **Simulation Control:** Setting up and controlling simulations are important features when using a fluid simulation system in computer games. Although the goal of this project is not to design an entire simulation system, ability to setup and control animations should be considered.

Having defined the goal of this project, we are now ready to begin the analysis.

Chapter 7

Discretization

To solve the Navier-Stokes equations using a computer, the equations have to be discretized, so numerical methods can be applied. To apply the numerical methods efficiently, the values of the fields representing velocity, pressure, temperature, etc. should be easy obtainable in the points needed by the numerical methods. This indicates that the discretization of the equations and the discretization of the vector fields are tightly connected, and should thus be considered together. In this chapter, some of the most commonly used discretization methods will be described.

In the CFD literature, many different types of discretizations have been presented, but the ones that are fit for the purpose of computer graphics narrows down to a couple. We will limit the detailed description to the two different discretizations used in [10, 12, 13, 14, 31, 32], since these are the most relevant for this project. In section 7.4, some examples of more complex grids will be given, to put the relevant grids into perspective.

7.1 Uniform Cartesian Grids in General

In this section, we will introduce the concept of *uniform Cartisian grids*, which covers the grids most often used for computer graphics applications. Both of the two grids, which we will consider for discretizing the vector fields in this project, are in this category.

That a grid is *cartesian* means that cells are axis-aligned boxes. Together, all the boxes form giant box, thus, we will think of a uniform cartesian grid as volume, placed in the all-positive octant of a three-dimensional left-handed coordinate system, divided into smaller cells. The front lower left cell is placed with one corner in origo.

The *uniform* property is defined differently in different contexts, so we will start by defining our meaning of the word. When this has been defined, we will then introduce our general notation for indexing uniform Cartesian grids, which we will need when using the grids for simulating fluid.

7.1.1 Uniformity

In some texts, uniformity is used to express that all cells cover the same volume. In a cartesian grid $\mathcal{G}$ this adapts to

$$\begin{aligned} \Delta x_{c_1} &= \Delta x_{c_2}, \\ \Delta y_{c_1} &= \Delta y_{c_2}, \\ \Delta z_{c_1} &= \Delta z_{c_2}, \quad \forall c_1, c_2 \in \mathcal{G}, \end{aligned} \tag{7.1}$$

where Δx_c, Δy_c, and Δz_c denote the size of cell c along the x axis, the y axis, and the z axis, respectively.

Other texts use the term uniform both to express that cells have the same size, i.e. uphold (7.1), *and* to restrict cells to be cubic, i.e.

$$\Delta x_c = \Delta y_c = \Delta z_c = h, \quad \forall c \in \mathcal{G}, \tag{7.2}$$

where h is refered to as the spatial discretization.

In some simulation environments, differences in width, height, and depth of cells could probably be used to slightly adapt the grid to the simulation environment. For instance, if simulating the draft through a long corridor, the resolution along the corridor would probably not have to be as high as the resolution across the corridor. However, having non-cubic cells could also disturb the symmetry of the simulation, causing rolling vortices to be "streched". More important, deviation from (7.2) affects the discretization of the Navier-Stokes equations described in chapter 8, making it more complex and thus slower to compute and harder to implement. Hence, we will assume throughout this report that uniformity covers both (7.1) and (7.2).

It should, however, still be possible to simulate non-cubic environments, so we will not restrict the entire grid to be cubic. Thus, the number of cells in the width, height, and depth of the grid need not be the same.

7.1.2 Indexing Uniform Cartesian Grids

With the properties defined above, uniform cartesian grids are very easy to work with. If $\mathcal{G}$ is a uniform cartesian grid with width W, height H, and depth D, a cell $c \in \mathcal{G}$ can be indexed by

$$c = c_{i,j,k}, \quad i, j, k \in \mathbb{Z}, \tag{7.3}$$

where

$$\begin{aligned} 0 \leq i < W, \\ 0 \leq j < H, \\ 0 \leq k < D. \end{aligned} \tag{7.4}$$

The coordinates of the center $\mathcal{C}$ of the cell c can be computed as

$$\mathcal{C}(c_{i,j,k}) = \begin{pmatrix} h/2 \\ h/2 \\ h/2 \end{pmatrix} + \begin{pmatrix} i \cdot h \\ j \cdot h \\ k \cdot h \end{pmatrix}, \quad \forall c_{i,j,k} \in \mathcal{G}, \tag{7.5}$$

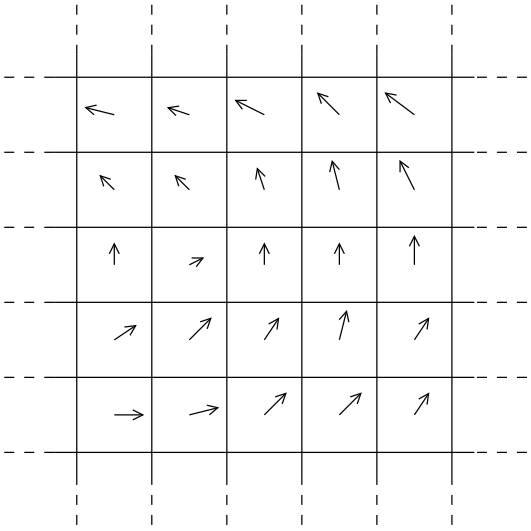


Figure 7.1: Collocated discretization. All fields (including the velocity field) are represented in the center of the cells.

where h is the width, height and depth of the cells.

For use in the discrete differentiation, the left, right, upper, lower, front, and back neighbors of a cell $c_{i,j,k} \in \mathcal{G}$ are simply defined, respectively, by

$$\begin{aligned}
\texttt{left}(c_{i,j,k}) &= c_{i-1,j,k}, \quad i > 0, \\
\texttt{right}(c_{i,j,k}) &= c_{i+1,j,k}, \quad i < W - 1, \\
\texttt{lower}(c_{i,j,k}) &= c_{i,j-1,k}, \quad j > 0, \\
\texttt{upper}(c_{i,j,k}) &= c_{i,j+1,k}, \quad j < H - 1, \\
\texttt{front}(c_{i,j,k}) &= c_{i,j,k-1}, \quad k > 0, \\
\texttt{back}(c_{i,j,k}) &= c_{i,j,k+1}, \quad k < D - 1.
\end{aligned} \tag{7.6}$$

It should be noted, that cells on the boundary of the grid only have five neighbors defined, e.g., the cells $c_{0,j,k} \in \mathcal{G}$ have no left neighbor defined in $\mathcal{G}$. Likewise, the six corner cells only have three neighbors.

7.2 Collocated Grids

In [31, 32], a special type of uniform cartesian grid is used, called a *collocated grid*. In a collocated grid, all fields in the simulation are represented at the center of cells, see figure 7.1. I.e., if

$$\mathbf{p} = \mathcal{C}(c_{i,j,k}), \quad c_{i,j,k} \in \mathcal{G},$$

then

$$\mathbf{u}(\mathbf{p}) = \mathbf{u}(\mathcal{C}c_{i,j,k})$$

is represented explicitly in the discretization of the field $\mathbf{u}$ with the grid $\mathcal{G}$. Values at other positions must be interpolated from the nearest values. To shorten the notation, we will simply use the $\mathbf{u}_{i,j,k}$ to specify

$$\mathbf{u}_{i,j,k} = \mathbf{u}(\mathcal{C}(c_{i,j,k})),\ c_{i,j,k} \in \mathcal{G}. \tag{7.7}$$

Although this representation is attractive because of its simplicity, unfortunately it has some numerical disadvantages. For instance, when using central differencing (see appendix A), on a collocated grid, the divergence $\nabla \cdot \mathbf{u}$ of the velocity field $\mathbf{u}$ is computed discretely as

$$\begin{aligned} \nabla \cdot \mathbf{u}_{i,j,k} &= \frac{\partial u^x_{i,j,k}}{\partial x} + \frac{\partial u^y_{i,j,k}}{\partial y} + \frac{\partial u^z_{i,j,k}}{\partial z} \\ &= \frac{u^x_{i+1,j,k} - u^x_{i-1,j,k} + u^y_{i,j+1,k} - u^y_{i,j-1,k} + u^z_{i,j,k+1} - u^z_{i,j,k-1}}{2h} \end{aligned} \tag{7.8}$$

I.e., the divergence in point $\mathbf{p} = \mathcal{C}(c_{i,j,k})$ is computed without regards to the value $\mathbf{u}(\mathbf{p})$ of the velocity field in that position. This is a general problem, since a stepwise linear function is not differentiable in the end points of the linear intervals. In this case it actually corresponds to taking the average of the divergences computed in the center of the neighboring intervals, since e.g.

$$\begin{aligned} \frac{1}{2}\left(\frac{\partial u^x_{i+1/2,j,k}}{\partial x} + \frac{\partial u^x_{i-1/2,j,k}}{\partial x}\right) &= \frac{1}{2}\left(\frac{u^x_{i+1,j,k} - u^x_{i,j,k}}{h} + \frac{u^x_{i,j,k} - u^x_{i-1,j,k}}{h}\right) \\ &= \frac{1}{2h}\left(u^x_{i+1,j,k} - u^x_{i-1,j,k}\right). \end{aligned} \tag{7.9}$$

This introduces numerical dissipation of the fluid motion, making the fluid appear thicker than it should be.

7.3 Staggered Grids

In [10, 12, 13, 14], another type of uniform cartesian grid, called a *staggered grid*, is used. With this type of grid, the pressure field is still represented at the center of cells, but velocities are represented on the faces between cells, and could thus be thought of as indicating flow between cells, see figure 7.2. I.e., velocity values

$$u^x(\mathbf{p}_1),\ u^x(\mathbf{p}_2),\ u^y(\mathbf{p}_3),\ u^y(\mathbf{p}_4),\ u^z(\mathbf{p}_5),\ u^z(\mathbf{p}_6), \tag{7.10}$$

are represented explicitly in the grid $\mathcal{G}$, if and only if

$$\begin{aligned} \mathbf{p}_1 &= \mathcal{C}(c_{i,j,k}) - (h/2, 0, 0)^T, \\ \mathbf{p}_2 &= \mathcal{C}(c_{i,j,k}) + (h/2, 0, 0)^T, \\ \mathbf{p}_3 &= \mathcal{C}(c_{i,j,k}) - (0, h/2, 0)^T, \\ \mathbf{p}_4 &= \mathcal{C}(c_{i,j,k}) + (0, h/2, 0)^T, \\ \mathbf{p}_5 &= \mathcal{C}(c_{i,j,k}) - (0, 0, h/2)^T, \\ \mathbf{p}_6 &= \mathcal{C}(c_{i,j,k}) + (0, 0, h/2)^T, \quad c_{i,j,k} \in \mathcal{G}. \end{aligned} \tag{7.11}$$

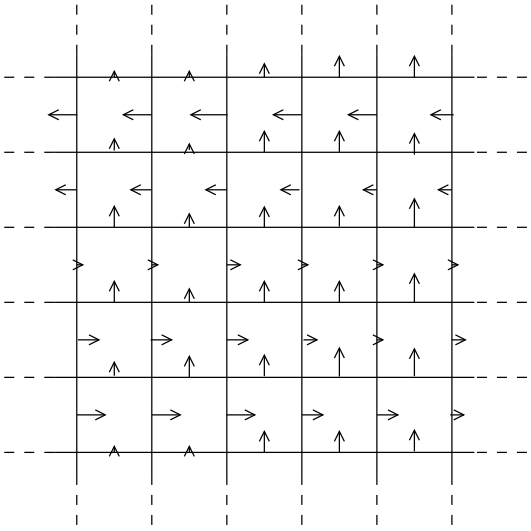


Figure 7.2: Staggered discretization. Velocities are represented at faces between cells. All other fields are represented in the center of cells.

Velocities at other positions are interpolated coordinatewise from the faces of the containing cell. Other values, such as pressures and forces, are still represented in the center of the cells. To shorten the notation we will use $u^x_{i-1/2,j,k}$ to specify

$$u^x_{i-1/2,j,k} = u^x\left(\mathcal{C}(c_{i,j,k}) - (h/2,0,0)^T\right), \tag{7.12}$$

and similarly for the values on the other five faces of cell $c_{i,j,k} \in \mathcal{G}$.

By representing velocity values in this way, the problem of numerical dissipation is minimized, since the central differencing in the center of a cell now yields

$$\begin{aligned} \nabla \cdot \mathbf{u}_{i,j,k} &= \frac{\partial u^x_{i,j,k}}{\partial x} + \frac{\partial u^y_{i,j,k}}{\partial y} + \frac{\partial u^z_{i,j,k}}{\partial z} \\ &= \frac{u^x_{i+1/2,j,k} - u^x_{i-1/2,j,k} + u^y_{i,j+1/2,k} - u^y_{i,j-1/2,k} + u^z_{i,j,k+1/2} - u^z_{i,j,k-1/2}}{h}. \end{aligned} \tag{7.13}$$

As we shall see, the updating of the fluid involves frequent use of velocity vectors. Since the velocity components are not represented in the same positions, unfortunately, these velocity vectors always have to be interpolated, thus, giving rise to numerical dissipation. Also, the update of the three velocity components will sometimes require three separate computations, since the velocity vectors are different at the positions of the three components, compared to just one in the collocated case.

Since forces are still represented in the center of cells, applying them to the velocity field also requires interpolation. The numerical dissipation introduced by this, however, is secondary, because it does not dissipate the velocity field directly.

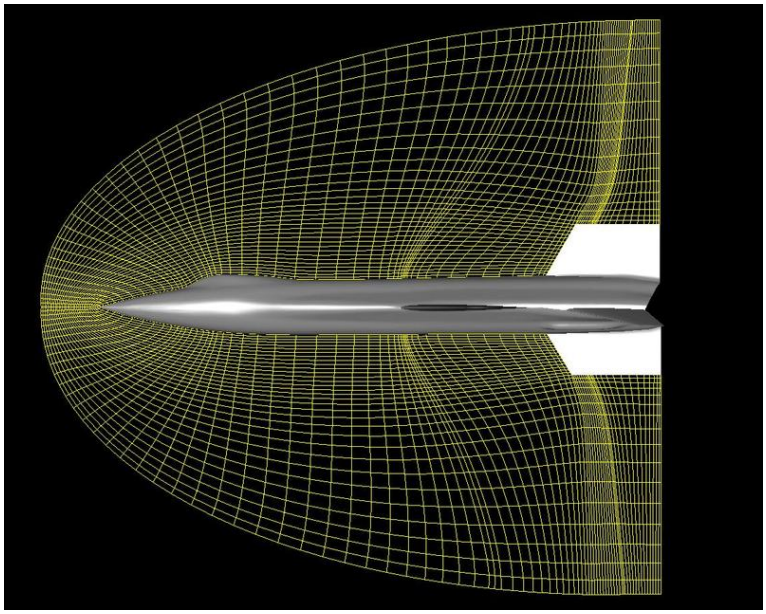


Figure 7.3: A structured grid fitted around the body of an X-150 aircraft. The image is taken from [25].

7.4 Other Grids

In the field of CFD many different types of grids are used for discretizing the simulation volume. Many of these grid are outside of the scope of this thesis, however, two classes of grids are especially interesting in this context.

7.4.1 Structured Grids

The uniform cartesian grids are the simplest examples of *structured grids*. The class of structured grids cover all grids that can be transformed into uniform cartesian grids, i.e., in 3D all cells have six faces and exactly six neighbors – one for each face of the cell. An example of a structured grid is shown in figure 7.3.

The advantage of structured grids is that they can model a scene very precisely. Thus, they are often used for analytic simulation, such as the examination of turbulent flow around an aircraft. At the same time, structured grids can be transformed into uniform cartesian grids, thus, performing a computation on the structured grid is similar to performing a computation on a uniform cartesian grid, but for some details, such as the spatial discretization factor. This implies that choosing a uniform cartesian grid for our solver is not a great restriction, since the solver will possibly be extendable to cover any type of structured grid.

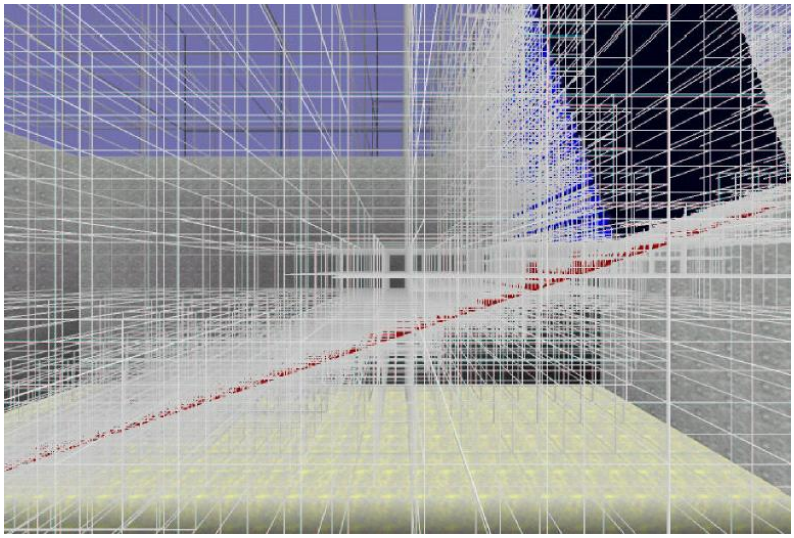


Figure 7.4: When discretizing the simulation volume using a balanced octree grid, high resolution can be achieved near objects, while keeping a low resolution in areas, where less detail is required.

7.4.2 Adaptive Grids

When modeling a scene with a uniform grid, the dilemma of choosing a resolution is encountered. The resolution must be chosen high enough for the objects in the scene to be modeled with a certain accuracy, but at the same time so low that the simulation does not take up too many resource and too much time – this is even more important, when discussing real-time simulation. It is thus very desirable that the grid can be changed, so high resolution is obtained where needed, while the number of cells are lowered in areas where less detail is required.

Structured grids, as mentioned above, can be adapted to a specific scene. However, changing a structured grid to reflect the motion of an object is a very complex task. Thus, structured grids are generally impractical for the simulation of moving objects. In [8], a grid based on a balanced octree structure [5] is used for the simulation of liquid, greatly lowering the number of cells used to represent the scene, see figure 7.4.

Unfortunately, the representation of adaptive grids for GPU implementation, is not as straight forward, as the representation of uniform cartesian grids – even for such geometrically simple grids as balanced octree grids. Thus, this is merely mentioned as a suggestion for possible future extension.

7.5 Choice of Discretization

Since the grids covered in section 7.4 are not easily representable with texture maps, we will disregard these in our choice of discretization. They are mentioned purely for putting the relevant methods into perspective.

Thus, when choosing the discretization for use in our solver, we will consider the options of *collocated* and *staggered* discretization. The following issues are considered in this decision:

- **Numerical Stability:** Both discretizations suffer from numerical dissipation, but in two different ways. Choosing the one discretization over the other on this account, requires better knowledge of the consequences of each type of numerical dissipation. Since we are not perfectly familiar with these consequences, we will not use this criteria in our choice.

- **Speed:** The speed of the solver is of great importance in this project, thus, we should consider the computational need of the two discretizations. It seems that the collocated discretization is better fit than the staggered, since velocity vectors are represented explicitly in the points where needed, and thus do not have to be interpolated in the same extend. The interpolation does seem simpler in the staggered discretization, since each component is only interpolated between two values. However, we fear that this is by far overshadowed by the extra use of the interpolation, as well as the extra work of updating three components separately instead of just one vector.

- **Simplicity and Intuition:** Building a fluid solver is a complex task. Many things depend on each other, and have to fit perfectly for the solver to work properly. The collocated discretization is much more simple and intuitive than staggered discretization. Especially when considering the representation with texture maps, which is necessary for our implementation.

- **Interaction with Objects:** For the addition of interaction with objects, the staggered discretization seems best fit. As described in chapter 9, the collocated discretization has some limitations on this subject.

On this basis, we will choose the collocated discretization over the staggered, since we weight the simplicity and speed stronger than the possibility of interaction with objects. We will, however, keep the staggered representation in mind as a future possibility. A comparison of the two methods, to clarify the differences, would be interesting.

Chapter 8

Updating the Velocity Field

In order for the simulation to be dynamic, the velocity field has to be updated regularly. Given a velocity field $\mathbf{u}^t$ at a given point in time t and the duration of time Δt, the task is to calculate a velocity field $\mathbf{u}^{t+\Delta t}$ for the time $t + \Delta t$, with regards to the Navier-Stokes equations and the forces working on the fluid.

Due to the complexity of the equations, the process of solving them numerically is not an easy task – even with the help from a powerful computer. To simplify the process, the solution is split into smaller steps. These steps can then be solved in succession, to yield the entirely updated velocity field.

In this chapter, the different steps involved in the update of the velocity field will be explained. The methods described are based on those of [10, 13, 31]. We will end the chapter, by giving an overview of the total solver.

To simplify the notation in the following sections, the known velocity field $\mathbf{u}^t$ is simply written as $\mathbf{u}$, and the updated (or yet unknown) velocity field $\mathbf{u}^{t+\Delta t}$ is written as $\tilde{\mathbf{u}}$.

Since we have already chosen to use a collocated representation, as described in chapter 7, the following description will be based on that. To encourage future implementation of a staggered representation, for instance to compare the two models, some of the important differences are summed up in section 8.7.

For each step, we will explain how to update a single cell in the simulation grid. The entire grid is updated with regards to a step by traversing the cells in the grid. Since many of the numerical methods are not defined consistently for the boundary cells, only the interior cells

$$c_{i,j,k} \in \mathcal{G}, \quad 0 < i < W - 1, \quad 0 < j < H - 1, \quad 0 < k < D - 1 \tag{8.1}$$

are traversed, where W, H, and D are the width, height, and depth of the grid $\mathcal{G}$, respectivly. How to update the boundary cells will be explained in the next chapter.

8.1 External Forces

The most simple step of updating the velocity field is the addition of external forces. In [31, 32], this is done in the simple manner

$$\tilde{\mathbf{u}}^f_{i,j,k} = \mathbf{u}_{i,j,k} + \Delta t \cdot \mathbf{f}_{i,j,k}, \tag{8.2}$$

where $\tilde{\mathbf{u}}^f$ is the velocity field after the addition of the external force field $\mathbf{f}$ to the velocity field $\mathbf{u}$. In sections 8.5 and 8.6, two examples of external forces are given.

8.2 Advection

The primary movement in the velocity field is described by the advective term of the Navier-Stokes equations, $-(\mathbf{u} \cdot \nabla)\mathbf{u}$, which describe advection of the velocity field by it self. For our solver, we will consider the following two methods for computing advection.

8.2.1 Finite Differencing

In [13], the advective contribution is computed using first-order central differencing (see appendix A). On a uniform collocated grid this yields

$$\begin{aligned} \Delta u^x_{i,j,k} &= \Delta t \Bigg(u^x_{i,j,k} \frac{u^x_{i+1,j,k} - u^x_{i-1,j,k}}{2h} + \\ & \quad u^y_{i,j,k} \frac{u^x_{i,j+1,k} - u^x_{i,j-1,k}}{2h} + \\ & \quad u^z_{i,j,k} \frac{u^x_{i,j,k+1} - u^x_{i,j,k-1}}{2h} \Bigg), \end{aligned} \tag{8.3}$$

$$\begin{aligned} \Delta u^y_{i,j,k} &= \Delta t \Bigg(u^x_{i,j,k} \frac{u^y_{i+1,j,k} - u^y_{i-1,j,k}}{2h} + \\ & \quad u^y_{i,j,k} \frac{u^y_{i,j+1,k} - u^y_{i,j-1,k}}{2h} + \\ & \quad u^z_{i,j,k} \frac{u^y_{i,j,k+1} - u^y_{i,j,k-1}}{2h} \Bigg), \end{aligned} \tag{8.4}$$

and

$$\begin{aligned} \Delta u^z_{i,j,k} &= \Delta t \Bigg(u^x_{i,j,k} \frac{u^z_{i+1,j,k} - u^z_{i-1,j,k}}{2h} + \\ & \quad u^y_{i,j,k} \frac{u^z_{i,j+1,k} - u^z_{i,j-1,k}}{2h} + \\ & \quad u^z_{i,j,k} \frac{u^z_{i,j,k+1} - u^z_{i,j,k-1}}{2h} \Bigg), \end{aligned} \tag{8.5}$$

where u^x, u^y, and u^z are the velocity components and h is the spatial discretization of the grid. The velocity field is then updated with regards to advection by the first-order time step

$$\tilde{\mathbf{u}}^a_{i,j,k} = \mathbf{u}_{i,j,k} + \Delta \mathbf{u}^a_{i,j,k}, \tag{8.6}$$

where $\Delta \mathbf{u}^a = (\Delta u^x_{i,j,k}, \Delta u^y_{i,j,k}, \Delta u^z_{i,j,k})^T$ is the advective contribution given by (8.3), (8.4), and (8.5), and $\tilde{\mathbf{u}}^a$ is the advected velocity field.

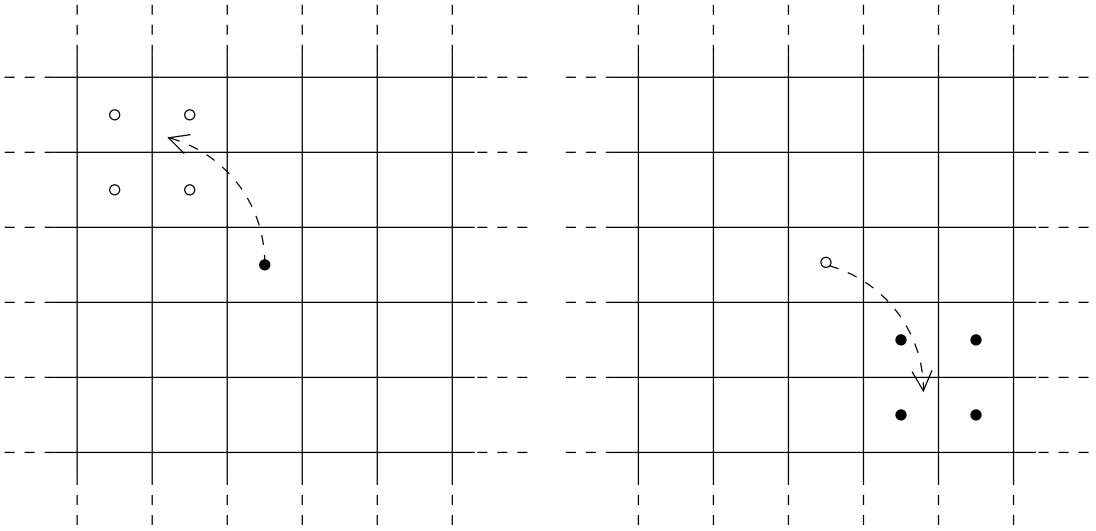


Figure 8.1: Lagrangian and semi-Lagrangian methods for advection. (a) With the Lagrangian method, values are moved forward. (b) With the semi-Lagrangian method, points are traced backwards, and the new values are found by interpolation. Solid circles indicate known velocities, and outlined circles indicate velocities to be updated.

Although this method is very suitable for fragment shader implementation, it is an explicit method and is thus unstable for large time steps. In fact, in order for this method to be stable, the CFL condition [13]

$$\max\left(u^x_{i,j,k}\frac{\Delta t}{\Delta x}, u^y_{i,j,k}\frac{\Delta t}{\Delta y}, u^z_{i,j,k}\frac{\Delta t}{\Delta z}\right) < 1, \tag{8.7}$$

must be upheld for all cells $c_{i,j,k} \in \mathcal{G}$, which means that no point in any cell in the grid, may be advected further than into a neighboring cell (including cells that share corners).

8.2.2 Semi-Lagrangian Method

Since advection intuitively is the force of the fluid movement acting on the fluid it self, the values of the velocity field should be moved forward in the direction given by the velocity vectors. This procedure is referred to as a Lagrangian method and is illustrated in figure 8.1 (a).

Due to the discretization of the grid, however, the Lagrangian method is impractical, since it is only possible to change the field values at certain points. In [31], Stam introduce a *semi-Lagrangian* method for advection, which is unconditionally stable. The idea of the semi-Lagrangian method is to advect a point $\mathbf{p}$ backwards in time, along its path $\mathcal{P}$ in the velocity field, yielding the point

$$\mathbf{p}^{t-\Delta t} = \mathcal{P}(\mathbf{p}, -\Delta t). \tag{8.8}$$

This is the point that advected forward yields the point $\mathbf{p}$. Thus, to advect the velocity field forward, the updated velocity $\tilde{\mathbf{u}}^a_{i,j,k}$ in the center of cell $c_{i,j,k}$ should be

$$\tilde{\mathbf{u}}^a_{i,j,k} = \mathbf{u}(\mathcal{P}(\mathcal{C}(c_{i,j,k}), -\Delta t)). \tag{8.9}$$

In [31], the point $\mathcal{P}(\mathcal{C}(c_{i,j,k}), -\Delta t)$ is approximated by dividing the time step Δt into the n equally sized steps

$$\delta t = \frac{\Delta t}{n}. \tag{8.10}$$

By using

$$\mathbf{p}_0 = \mathcal{C}(c_{i,j,k}) \tag{8.11}$$

as the starting point, the ending point

$$\mathbf{p}_n \approx \mathcal{P}(\mathcal{C}(c_{i,j,k}), -\Delta t) \tag{8.12}$$

can be computed by

$$\mathbf{p}_i = \mathbf{p}_{i-1} - \delta t \cdot \mathbf{u}(\mathbf{p}_{i-1}). \tag{8.13}$$

The intermediate velocity $\mathbf{u}(\mathbf{p}_{i-1})$ is computed in every step by linear interpolation between the nearest grid points, as is the final updated velocity value $\mathbf{u}(\mathcal{P}(\mathcal{C}(c_{i,j,k}), -\Delta t))$. In this way, updated velocity values are only computed at grid points, as illustrated in figure 8.1 (b). In [32], good results are achieved using only one step, simplifying the calculations to

$$\tilde{\mathbf{u}}^a_{i,j,k} = \mathbf{u}(\mathcal{C}(c_{i,j,k}) - \Delta t \mathbf{u}_{i,j,k}), \tag{8.14}$$

and in [17], this simple version has been implemented to run on GPUs.

Though stable for big time steps, this method suffers from volume-loss. This means that it cannot be used for advecting fields where high precision is needed, such as water surfaces. However, when used in simulations of diffusive substances with no free surface, such as smoke, the loss of volume appear natural.

8.3 Diffusion

The diffusion term of the Navier-Stokes equations, $\nu\nabla^2\mathbf{u}$, describes the effect of mass being exchanged between neighboring cells, as illustrated in figure 8.2. For the simulation of diffusion, we consider the following two methods.

8.3.1 Simple Diffusion

In [13], as with advection, diffusion is added by using first-order central differencing directly on the diffusion term. On a uniform collocated grid this yields

$$\begin{aligned} \Delta\mathbf{u}^d_{i,j,k} &= \Delta t\ \nu\ \frac{1}{h^2}\Big(\mathbf{u}_{i-1,j,k} + \mathbf{u}_{i+1,j,k} + \\ &\quad \mathbf{u}_{i,j-1,k} + \mathbf{u}_{i,j+1,k} + \mathbf{u}_{i,j,k-1} + \mathbf{u}_{i,j,k+1} - 6\mathbf{u}_{i,j,k}\Big), \end{aligned} \tag{8.15}$$

where ν is the kinematic viscosity and h is the spatial discretization.

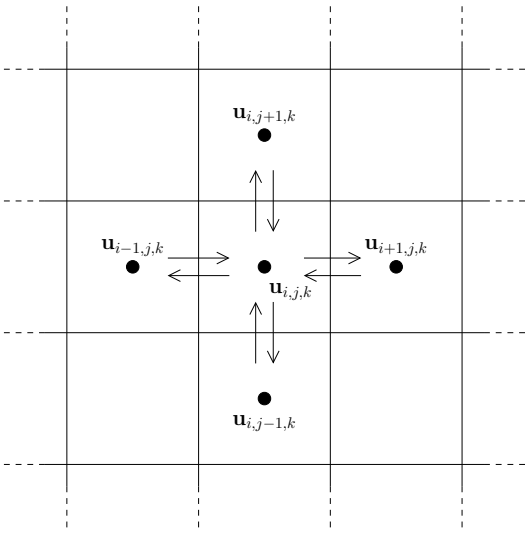


Figure 8.2: Diffusion: Mass is exchanged between neighboring cells.

When $\Delta\mathbf{u}^d$ has been computed, the velocity field is updated, with regards to diffusion, by the first-order time step

$$\tilde{\mathbf{u}}^d_{i,j,k} = \mathbf{u}_{i,j,k} + \Delta\mathbf{u}^d_{i,j,k}. \tag{8.16}$$

This method is very suitable for fragment shader implementation, but because it is an explicit method, it is unstable for large time steps and for large values of ν.

8.3.2 Stable Diffusion

In [31], diffusion is solved implicitly, making the solution stable. I.e., instead of solving (8.16) by computing $\Delta\mathbf{u}^d$ from (8.15), the diffused velocity field is found by solving

$$\mathbf{u} = \tilde{\mathbf{u}}^d - \Delta\tilde{\mathbf{u}}^d, \tag{8.17}$$

for the unknown velocity field $\tilde{\mathbf{u}}^d$. Intuitively, this could be thought of as finding the velocity field, which diffused backwards yields the currently known velocity field.

In [32], this diffusion scheme is solved using a simple *Gauss-Seidel method* [6]. Since the Gauss-Siedel method cannot be implemented on a GPU, however, the Jacobi method [6, 9] is used instead.

Discretizing (8.17) using first-order central differences yields

$$\begin{aligned}
\mathbf{u}_{i,j,k} &= \tilde{\mathbf{u}}^d_{i,j,k} - \Delta\tilde{\mathbf{u}}^d_{i,j,k} \\
&= \tilde{\mathbf{u}}^d_{i,j,k} - \frac{\Delta t\ \nu}{h^2}\Big(\tilde{\mathbf{u}}^d_{i-1,j,k} + \tilde{\mathbf{u}}^d_{i+1,j,k} + \\
&\quad \tilde{\mathbf{u}}^d_{i,j-1,k} + \tilde{\mathbf{u}}^d_{i,j+1,k} + \tilde{\mathbf{u}}^d_{i,j,k-1} + \tilde{\mathbf{u}}^d_{i,j,k+1} - 6\tilde{\mathbf{u}}^d_{i,j,k}\Big)
\end{aligned} \tag{8.18}$$

By isolating the unknown velocity $\tilde{\mathbf{u}}^d_{i,j,k}$ we get

$$\begin{aligned}\tilde{\mathbf{u}}^d_{i,j,k} &= \frac{\mathbf{u}_{i,j,k}}{1+6a} + \frac{a}{1+6a}\Big(\tilde{\mathbf{u}}^d_{i-1,j,k} + \tilde{\mathbf{u}}^d_{i+1,j,k} + \\ &\quad \tilde{\mathbf{u}}^d_{i,j-1,k} + \tilde{\mathbf{u}}^d_{i,j+1,k} + \tilde{\mathbf{u}}^d_{i,j,k-1} + \tilde{\mathbf{u}}^d_{i,j,k+1}\Big),\end{aligned} \tag{8.19}$$

where $a = \frac{\Delta t\ \nu}{h^2}$. From this, the Jacobi steps can be formed as (see appendix B)

$$\tilde{\mathbf{u}}^{(0)}_{i,j,k} = \mathbf{0}, \tag{8.20}$$

$$\begin{aligned}\tilde{\mathbf{u}}^{(m)}_{i,j,k} &= \frac{\mathbf{u}_{i,j,k}}{1+6a} + \frac{a}{1+6a}\Big(\tilde{\mathbf{u}}^{(m-1)}_{i-1,j,k} + \tilde{\mathbf{u}}^{(m-1)}_{i+1,j,k} + \\ &\quad \tilde{\mathbf{u}}^{(m-1)}_{i,j-1,k} + \tilde{\mathbf{u}}^{(m-1)}_{i,j+1,k} + \tilde{\mathbf{u}}^{(m-1)}_{i,j,k-1} + \tilde{\mathbf{u}}^{(m-1)}_{i,j,k+1}\Big), \quad m > 0.\end{aligned} \tag{8.21}$$

8.4 Mass-Conservation

When the velocity field has been updated with advection and diffusion, it does not necessarily uphold the incompressibility condition, $\nabla \cdot \mathbf{u} = 0$. For the flow to appear natural, and have naturally looking swirls, the velocity field needs to be corrected to be mass-conserving. This is done by adding the pressure term of the Navier-Stokes equations.

We have

$$\tilde{\mathbf{u}} = \mathbf{u} - \frac{1}{\rho}\nabla p, \tag{8.22}$$

where $\tilde{\mathbf{u}}$ is the fully updated, mass-conserving velocity field, $\mathbf{u}$ is the velocity field after the above steps, and $-\frac{1}{\rho}\nabla p$ is the pressure term of the Navier-Stokes equations. However, since incompressibility states that $\nabla \cdot \tilde{\mathbf{u}} = 0$, we have

$$\begin{aligned}\nabla \cdot \tilde{\mathbf{u}} &= \nabla \cdot \left(\mathbf{u} - \frac{1}{\rho}\nabla p\right) \\ &= \nabla \cdot \mathbf{u} - \frac{1}{\rho}\nabla^2 p \\ &= 0 \\ \iff \quad \frac{1}{\rho}\nabla^2 p &= \nabla \cdot \mathbf{u}.\end{aligned} \tag{8.23}$$

Equation (8.23) is known as a Poisson equation for the unknown pressure field p, and can be solved using a number of different methods. In [31], a multigrid method is used, [10, 12] use a *Conjugate Gradient* method, and in [32] the simpler Gauss-Seidel method is used. For an introduction of these methods, we refer to [6]. However, none of these methods are well suited for implementation on a GPU. In [17], Harris et al. have successfully implemented different variations of the Jacobi method (again see [6]) on a GPU.

The RHS of equation (8.23) is the *divergence* of $\mathbf{u}$. When discreticed using first-order central differencing on a collocated grid, we get

$$D_{i,j,k} = \frac{u^x_{i+1,j,k} - u^x_{i-1,j,k} + u^y_{i,j+1,k} - u^y_{i,j-1,k} + u^z_{i,j,k+1} - u^z_{i,j,k-1}}{2h} \tag{8.24}$$

Discretizing the LHS in the same way yields

$$\frac{p_{i-1,j,k} + p_{i+1,j,k} + p_{i,j-1,k} + p_{i,j+1,k} + p_{i,j,k-1} + p_{i,j,k+1} - 6p_{i,j,k}}{\rho h^2} = D_{i,j,k}. \tag{8.25}$$

By isolating the unknown pressure $p_{i,j,k}$ we get

$$p_{i,j,k} = \frac{1}{6}\Big(p_{i-1,j,k} + p_{i+1,j,k} + p_{i,j-1,k} + p_{i,j+1,k} + p_{i,j,k-1} + p_{i,j,k+1} - \rho h^2 D_{i,j,k}\Big). \tag{8.26}$$

This implies that if we first compute the divergence field D by equation (8.24), the pressure field p can be computed iteratively using the Jacobi steps (see appendix B)

$$p^0 = 0, \tag{8.27}$$

$$p^{(m)}_{i,j,k} = \frac{1}{6}\Big(p^{(m-1)}_{i-1,j,k} + p^{m}_{i+1,j,k} + p^{(m-1)}_{i,j-1,k} + p^{(m-1)}_{i,j+1,k} + p^{(m-1)}_{i,j,k-1} + p^{(m-1)}_{i,j,k+1} - \rho h^2\ D_{i,j,k}\Big), \quad m > 0. \tag{8.28}$$

When the pressure field is computed the velocity field is adjusted according to (8.22), yielding a mass-conserving velocity field. On a collocated grid this is done by

$$\tilde{u}^x_{i,j,k} = u^x_{i,j,k} - \frac{1}{2\rho h}(p_{i+1,j,k} - p_{i-1,j,k}), \tag{8.29}$$

$$\tilde{u}^y_{i,j,k} = u^y_{i,j,k} - \frac{1}{2\rho h}(p_{i,j+1,k} - p_{i,j-1,k}), \tag{8.30}$$

$$\tilde{u}^z_{i,j,k} = u^z_{i,j,k} - \frac{1}{2\rho h}(p_{i,j,k+1} - p_{i,j,k-1}). \tag{8.31}$$

8.5 Vorticity Confinement

When simulating fluid on such relatively coarse grids as often used in computer graphics, the small scale detail in fluid movement is missing. The large scale details are present, but due to the coarseness of the grids these are damped by numerical dissipation, causing the fluid to move unnatural. In [10], Fedkiw et al. use a method called *vorticity confinement* [33], which adds this missing detail back into the simulation. The method work in the following three steps.

In the first step, the *vorticity*

$$\boldsymbol{\omega} = \nabla \times \mathbf{u}, \tag{8.32}$$

is computed for the velocity field $\mathbf{u}$. Each vorticity vector represents a local spinning in the flow, the direction being the axis of this spinning, and the length being the magnitude of the spinning.

Next, the vorticity vectors are used to compute vorticity location vectors

$$\mathbf{N} = \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|}, \quad \boldsymbol{\eta} = \nabla|\boldsymbol{\omega}|. \tag{8.33}$$

Vorticity location vectors are vectors pointing from areas of low vorticity to areas of high vorticity.

Finally, to add the missing detail back into the velocity field, the vorticity confinement force is computed as

$$\mathbf{f}_{conf} = \epsilon \; h \; (\mathbf{N} \times \boldsymbol{\omega}), \tag{8.34}$$

where $\epsilon > 0$ is a parameter used to control the magnitude of the vorticity confinement effect and h is the spatial discretization. Due to the cross product of $\mathbf{N}$ with $\boldsymbol{\omega}$, the confinement forces are tangential to the iso-surfaces of vorticity magnitude, and point in directions around local vortices.

Discretized using first-order central differences on a collocated grid, equations (8.32) and (8.33) yield

$$\boldsymbol{\omega}_{i,j,k} = \frac{1}{2h} \begin{pmatrix} u^z_{i,j+1,k} - u^z_{i,j-1,k} - u^y_{i,j,k+1} + u^y_{i,j,k-1} \\ u^x_{i,j,k+1} - u^x_{i,j,k-1} - u^z_{i+1,j,k} + u^z_{i-1,j,k} \\ u^y_{i+1,j,k} - u^y_{i-1,j,k} - u^x_{i,j+1,k} + u^x_{i,j-1,k} \end{pmatrix} \tag{8.35}$$

and

$$\boldsymbol{\eta}_{i,j,k} = \frac{1}{2h} \begin{pmatrix} |\boldsymbol{\omega}_{i+1,j,k}| - |\boldsymbol{\omega}_{i-1,j,k}| \\ |\boldsymbol{\omega}_{i,j+1,k}| - |\boldsymbol{\omega}_{i,j-1,k}| \\ |\boldsymbol{\omega}_{i,j,k+1}| - |\boldsymbol{\omega}_{i,j,k-1}| \end{pmatrix}. \tag{8.36}$$

When the confinement forces has been computed discretely, they can be added to the simulation as described in section 8.1.

8.6 Thermal Buoyancy

Another example of an external force, which is often used in the simulation of gases, is that of *thermal buoyancy*. This force is what makes hot air rise and cool air sink.

In [10], the force of thermal buoyancy is described as (simplified from [14])

$$\mathbf{f}_T = \beta \mathbf{y}(T - T_0), \tag{8.37}$$

where β is a positive coefficient that controls the magnitude of the thermal buoyancy, derived from the coefficient of thermal expansion, $\mathbf{y}$ is a vector pointing in the vertical direction, T_0 is the average temperature in the simulation volume, and T is the local temperature of the gas.

For the addition of thermal buoyancy, a representation of the scalar temperature field T is needed, and this field needs to be updated, so the temperature evolves correctly. In [14], evolution of the temperature field is modeled by the equation

$$\frac{\partial T}{\partial t} = \lambda \nabla^2 T - (\mathbf{u} \cdot \nabla) T, \tag{8.38}$$

where λ is a the thermal distribution coefficient. This equation is similar to the diffusive and advective terms of the Navier-Stokes equations, and can thus be solved using the same methods, as described in section 8.2 and 8.3.

When the thermal buoyancy forces has been computed, they are added to the simulation as described in section 8.1.

8.7 Updating a Staggered Representation

Many of the steps described above are similar when using a staggered representation, except for some differences in the discrete differentiation. However, some steps differ remarkably, and should thus be handled differently. In this section we will give a short description of the steps described above in the context of a staggered representation.

8.7.1 External Forces

Since external forces are center-based in either representation, the force field it self is handled in the same way. When forces are added to the velocity field, however, they have to be "converted" to face-based representation. This is done by simple averaging, yielding

$$\tilde{u}^x_{i-1/2,j,k} = u^x_{i-1/2,j,k} + \frac{\Delta t}{2}(f^x_{i-1,j,k} + f^x_{i,j,k}), \tag{8.39}$$

$$\tilde{u}^y_{i,j-1/2,k} = u^y_{i,j-1/2,k} + \frac{\Delta t}{2}(f^y_{i,j-1,k} + f^y_{i,j,k}), \tag{8.40}$$

$$\tilde{u}^z_{i,j,k-1/2} = u^z_{i,j,k-1/2} + \frac{\Delta t}{2}(f^z_{i,j,k-1} + f^z_{i,j,k}). \tag{8.41}$$

8.7.2 Advection

The options for advection of the staggered velocity field are the same as with the collocated velocity field.

In the case of finite differencing, the only difference between staggered and collocated representation is the way the advective term of the Navier-Stokes equations is discretized.

In the case of the semi-Lagrangian method, the difference is that instead of tracing back the center of the cells, the center of each face must be traced back separately, thus, requiring three advection steps per cell, instead of just one in the collocated case.

8.7.3 Diffusion

When discretizing the diffusion term of the Navier-Stokes equations, the x, y, and z components are separated, i.e., the x component can be computed without paying attention to the other components. Thus, diffusion can be added to the velocity field by diffusing each component of the field, as if it was a scalar field. This yields that a staggered representation is updated with regards to diffusion in same manner as a collocated representation, except for the displacement of the three components.

8.7.4 Mass-Conservation

Since the pressure field is center-based in either type of representation, the Poisson-equation (8.23) is solved in the exact same way. Thus, the only difference between collocated and staggered representation is that when velocities are face-based, the computation of the divergence and pressure gradients are numerically more precise, because the discrete differentiation fit better to the representation of the values. On a staggerede grid,

the divergence field $D = \nabla \cdot \mathbf{u}$ is computed discretely by

$$D_{i,j,k} = \frac{u^x_{i+1/2,j,k} - u^x_{i-1/2,j,k} + u^y_{i,j+1/2,k} - u^y_{i,j-1/2,k} + u^z_{i,j,k+1/2} - u^z_{i,j,k-1/2}}{h}, \tag{8.42}$$

and the velocities are adjusted according to equation (8.22) by

$$\tilde{u}^x_{i-1/2,j,k} = u^x_{i-1/2,j,k} - \frac{1}{h}(p_{i,j,k} - p_{i-1,j,k}), \tag{8.43}$$

$$\tilde{u}^y_{i,j-1/2,k} = u^y_{i,j-1/2,k} - \frac{1}{h}(p_{i,j,k} - p_{i,j-1,k}), \tag{8.44}$$

$$\tilde{u}^z_{i,j,k-1/2} = u^z_{i,j,k-1/2} - \frac{1}{h}(p_{i,j,k} - p_{i,j,k-1}). \tag{8.45}$$

8.7.5 Vorticity Confinement

Due to the involvement of the ∇-operator with the cross product, discretizing the vorticity vectors on a staggered grid, is impractical. Instead, [10] suggests that velocities are transformed to centered representation, by averaging componentwise, i.e.

$$\mathbf{u}_{i,j,k} = \frac{1}{2} \begin{pmatrix} u^x_{i-1/2,j,k} + u^x_{i+1/2,j,k} \\ u^y_{i,j-1/2,k} + u^y_{i,j+1/2,k} \\ u^z_{i,j,k-1/2} + u^z_{i,j,k+1/2} \end{pmatrix} \tag{8.46}$$

Center-based confinement forces are then computed in the same way as with the collocated representation and added to the velocity field, as described in section 8.7.1.

8.7.6 Thermal Buoyancy

Since the temperature and the force fields are both center-based in either representation, thermal buoyancy forces are calculated in the exact same way.

8.8 Method Overview

Having gone through all the steps involved, we are now ready to assemble the solver.

According to [10], the effect of viscosity in air is negligible, when simulated on a coarse grid. The kinematic viscosity of air is very low, so the numerical dissipation resulting from the coarse discretization, will often dominate the diffusive contribution. Since our purpose is to simulate smoke distributed in air, we will not include the diffusion step in our method.

Since the semi-Lagrangian method for advection is stable even for large time-steps, choosing this method is crucial for the purpose of computer games. For maximum speed, as in [32], we will only use a single step, when tracing back positions.

In order to obtain real-time simulation rates, we will presumably have to use very coarse simulation grids, even coarse than the grids used in [10]. Thus, vorticity confinement is considered highly relevant, in order to produce realistically swirling smoke.

The thermal buoyancy step is relatively uncomplicated. We will thus add this feature, since we believe it will improve the visual realism.

In [32], mass-conservation is enforced on the velocity field both before and after advection. This is done to diminish the volume-loss introduced by the semi-Lagrangian advection method. In order to speed up the simulation, we will spare one of these steps, so only one mass-conservation step is performed. Hence, we will order the steps of the solver, so the temperature and velocity fields can be advected using a mass-conserving velocity field, regardless. This might cause severe volume-loss, but hopefully it will create pleasing results.

The final assembly of the solver can thus be written as:

1. Advect the temperature field T (section 8.6)
2. Advect the velocity field $\mathbf{u}$ (section 8.2.2)
3. Add any user-supplied force fields (section 8.1)
4. Add the thermal buoyancy force f_T (section 8.6)
5. Add the vorticity confinement force f_{conf} (section 8.5)
6. Compute the pressure field p (section 8.4)
7. Adjust the velocity field $\mathbf{u}$ according to pressure p (section 8.4)

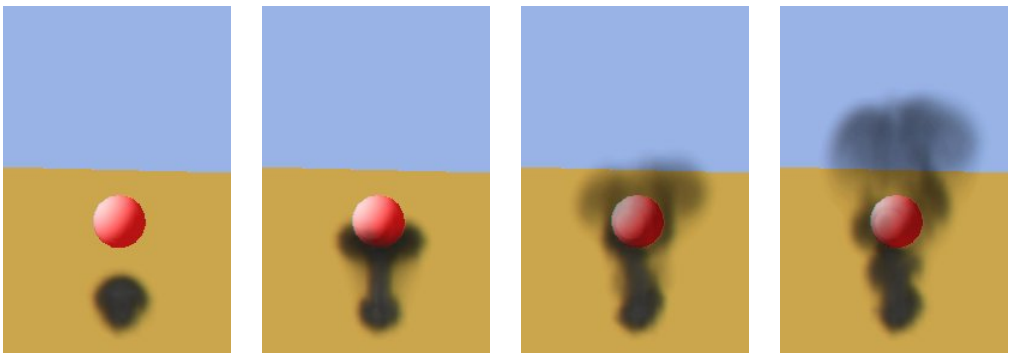
Chapter 9

Boundary Conditions

In order to create more advanced scenes than just swirling smoke, the fluid should be able to interact with objects in a scene. By introducing different types of *boundary conditions* for the differential equations solved in chapter 8, interaction with both stationary and moving objects can be simulated. An example of such interaction is shown in figure 9.1.

In this section, boundary conditions related to simple interaction with stationary objects are introduced, and we discuss some of the problems involved with upholding these boundary conditions.

Interaction with arbitrary geometric objects is often desired. However, to simplify the boundary conditions, we will limit the following description to objects fitting the grids mentioned in chapter 7. I.e., cells in the grid are either completely filled with fluid or fully occupied by an object.



(a) (b) (c) (d)

Figure 9.1: Example of interaction with a spherical object. The smoke flows around the object creating chaotic swirls.

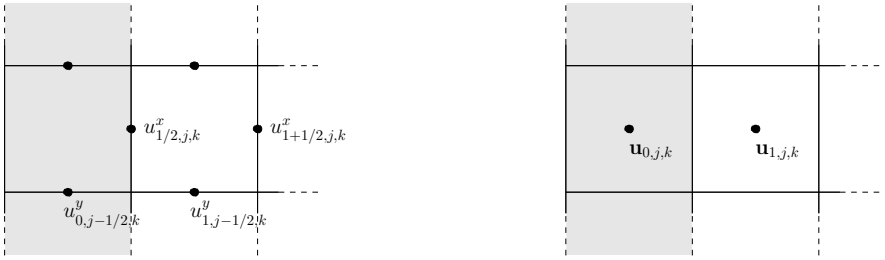


Figure 9.2: Boundary conditions for a wall boundary. (a) On a staggered grid the velocity component can be set to 0 directly. (b) On a collocated grid the velocity component has to be set to the inverse of the adjacent velocity. The boundary cells are indicated by a gray shade. Solid circles indicate the explicit velocity representations.

9.1 Boundaries of the Fluid Volume

In chapter 8, we explained how to update the velocity by traversing the interior cells of the fluid volume. For the numerical methods described in chapter 8 to be numerically stable, we need a set of boundary conditions, to control the behavior of the fluid in the boundary cells of the fluid volume, i.e. cells

$$c_{0,j,k},\ c_{W-1,j,k},\ c_{i,0,k},\ c_{i,H-1,k},\ c_{i,j,0},\ c_{i,j,D-1} \in \mathcal{G}, \tag{9.1}$$

where W, H, and D are the width, height, and depth of the grid $\mathcal{G}$, respectively.

9.1.1 Closed Volume

In [32], boundary conditions are described, for treating the simulation volume as a closed box, with no fluid entering or leaving the volume. I.e., the boundary cells can be thought of as walls. In general, this is upheld by keeping the velocity normal to the boundary steady at 0.

In a staggered grid, the velocity is explicitly defined at the boundary. This boundary condition can thus be enforced simply by setting the velocity component to 0. For the case shown in figure 9.2 (a), this would mean setting $u^x_{1/2,j,k} = 0$.

In a collocated grid, this is a little more complicated, since the normal velocities are not represented explicitly. However, since velocities are interpolated linearly we have, for the case shown in figure 9.2 (b),

$$u^x_{1/2,j,k} = \frac{u^x_{0,j,k} + u^x_{1,j,k}}{2} = 0 \quad \Longleftrightarrow \quad u^x_{0,j,k} = -u^x_{1,j,k}. \tag{9.2}$$

I.e., to uphold a normal velocity of 0, the according velocity component of the boundary cell should be set to the inverse of the according velocity component in the adjacent cell.

The boundary values should at all times reflect the values of the interior cells, so the mentioned boundary conditions need to be enforced every time the internal values are changed.

Free-Slip and Non-Slip Obstacles

In [13], the velocities tangential to the boundary are used to control the friction between the fluid and the bounding wall. Setting $u^y_{0,j-1/2,k}$ on figure 9.2 (a) to

$$u^y_{0,j-1/2,k} = a \cdot u^y_{1,j-1/2,k}, \quad a \in [-1, 1], \tag{9.3}$$

introduces the *friction coefficient* a, where $a = 1$ yields no friction (*free-slip*) and $a = -1$ yields a tangential velocity of 0 (*non-slip*).

9.1.2 Free Flow

If no interaction with the boundaries of the simulation volume should occur, i.e., fluid can flow freely in and out of the volume, the boundary velocities should appear as though they were part of the simulation. This can be done by setting the boundary values to the same as the adjacent values. I.e., for the staggered case shown in figure 9.2 (a)

$$\begin{aligned} u^x_{1/2,j,k} &= u^x_{1+1/2,j,k}, \\ u^y_{0,j-1/2,k} &= u^y_{1,j-1/2,k}, \end{aligned} \tag{9.4}$$

and for the collocated case in figure 9.2 (b)

$$\mathbf{u}_{0,j,k} = \mathbf{u}_{1,j,k}. \tag{9.5}$$

9.1.3 Boundary Conditions for the Non-Velocity Fields

As well as for the velocity field, proper boundary conditions have to be enforced for the other vector and scalar fields involved in the simulation. These boundary conditions are, however, independent of the type of boundary, and are thus handled similarly in either of the above mentioned cases.

Since differences in pressure should not cause acceleration across boundaries, the pressure in a boundary cell is set equal to the pressure in the adjacent internal cell, so $\nabla p = 0$. This is known as *pure Neumann* boundary conditions. To allow density and temperature to be conserved alongside boundaries, density and temperature values in boundary cells should be set equal to the values of the adjacent cells. To prevent boundaries from introducing extra vorticity, vorticity vectors in boundary cells should also be set to the same as in the adjacent cells.

9.1.4 Other Boundary Effects

The boundary conditions for the fluid volume can also be used actively, to create advanced boundary effects. In [13], sinusoidal boundary values are suggested for simulating waves in and out of a bay. Similarly, boundary values can be used for simulating vents and drafts.

9.2 Stationary Objects

The simplest form of fluid/object interaction is that of stationary objects, also referred to as *obstacles*. As with the boundaries of the fluid volume, we need to assure that the fluid does not enter the object by setting the velocity normal to the object surface to 0. Since we are only working with objects that are representable in the simulation grid, the face between a fluid-cell and an object-cell can be thought of as the surface of the object. Thus, to represent a solid object in the fluid, we need to set the normal velocity to 0, for all faces between object cells and fluid cells. Keeping boundary values updated allows us to solve the fluid steps, as described in chapter 8, without paying any attention to the locations of obstacles, and still get the effect of obstacle behavior.

9.2.1 Objects in a Staggered Grid

In the staggered model, setting boundary values is straight forward, and can be done in the same way as with the fluid volume boundaries described above.

Furthermore, velocities inside a cell are interpolated componentwise from the faces of the cell, i.e., in the cell $c_{i,j,k}$, u^x is interpolated from $u^x_{i-1/2,j,k}$ and $u^x_{i+1/2,j,k}$, etc. Thus, without any special considerations, objects in a staggered grid can actually be specified on face-level.

9.2.2 Objects in a Collocated Grid

In the collocated case, the enforcement of internal boundary conditions introduce some problems.

Problem With Corners

When setting the boundary conditions for a corner cell, in the way described for the fluid volume boundary in section 9.1, interpolation does not yield a normal velocity of 0 at the obstacle surface, see figure 9.3. This cannot be corrected simply by setting the boundary velocities, but have accounted for in the interpolation as well. However, it is not certain how severe the visual consequences of this will be, and since the voxel representation of objects is already a coarse approximation, extending the interpolation seems like overkill.

Problem With Thin Objects

A more severe problem is encountered if an obstacle is only one cell wide. If the velocities in two cells on opposite sides of the boundary have opposite directions, setting the boundary cell velocity cannot uphold a normal velocity of 0 for both sides of the object, see figure 9.4.

Possible Fix

An idea on how to get around these problems would be to simply set the velocity in obstacle cells to 0, even in the collocated grid. This is actually equivalent to using a

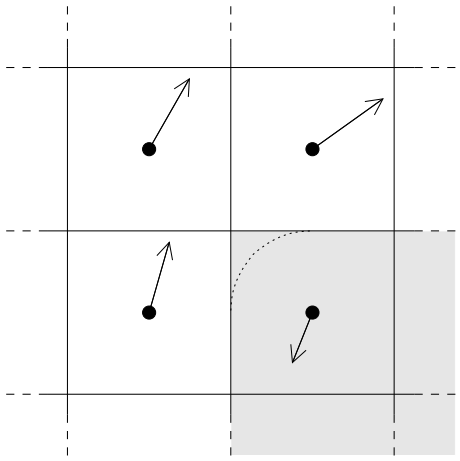


Figure 9.3: Setting the boundary values for a corner cell in collocated grid. Even though the velocity components are set accordingly, the corner is not upheld. The obstacle will thus appear to have a round corner, as indicated by the dotted line.

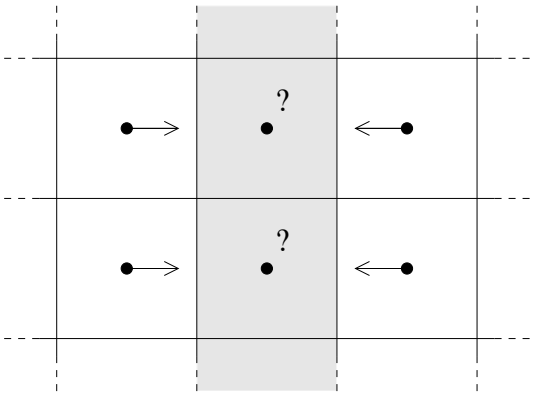


Figure 9.4: Problem with thin obstacles in a collocated grid. When an obstacle is only one cell wide, the boundary values cannot uphold both sides of the obstacle.

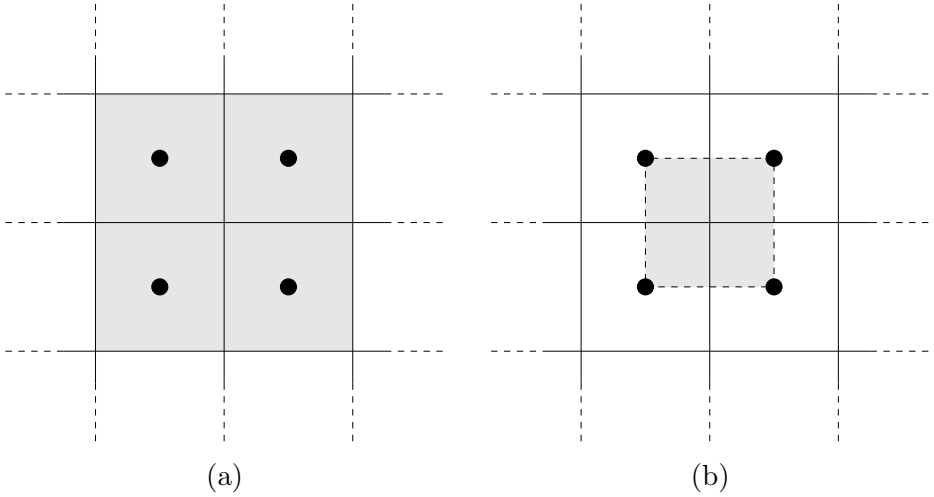


Figure 9.5: The effect of setting center-based velocities to 0. The shaded cells (a) indicate obstacle cells. When setting the velocities in the center of these cells to 0, the actual obstacle will appear as indicated with the shaded region in (b).

whole other grid, where velocities are represented in the corners of cells, since interpolated normal velocities of 0, hence, will occur from cell center to cell center, see figure 9.5.

This introduces the following "classes" of obstacles:

- **One cell:** Specifying an obstacle of only one cell would create a *point* obstacle.
- **Two cells:** An obstacle of two adjacent cells would create a *line* obstacle
- **Four cells:** Four adjacent cells (2×2) would create a plane, rectangular obstacle.
- **Eight cells:** Specifying eight obstacle cells ($2 \times 2 \times 2$) would create a box-shaped obstacle.

Although not as intuitive as the faces in the staggered grid, the lower classes of obstacles (points and lines) could maybe even open up for some special effects, such as fluid fix-points.

9.3 Choice of Boundaries

The volume boundaries, as explained in section 9.1, are easily implementable in fragment shaders. Furthermore, they are necessary for the solver to be stable, and can thus no be left out.

Internal boundaries are not as easily handled by fragment shaders, since the setting of these involves arbitrary references to neighbor cells. If the time allows us to experiment with internal boundaries, we will limit this to the simple, yet possibly not sufficient, method of setting velocities in obstacle cells to 0.

Chapter 10

Visualization

Although the visualization is not the primary focus of this thesis, some minimum extend of visualization is required to evaluate the described method. Furthermore, for the use of the method in computer games, visualization of the simulations would be an important issue. Thus, in this section we will describe some of the possibilities for visualizing the simulations, made with the described method.

10.1 Particle Systems

Due to their simplicity and scalability, particle systems are already widely used for simulating and rendering gases in computer games. Most modern game engines already support particle systems in an efficient and optimized manner. Thus, using particle systems for the purpose of this project seems logical.

When rendering fluids, such as water and liquids in general, where the surface is, in fact, defining the substance, particle system come short. Using particle systems for rendering liquid effects tends to make the surface of the liquid seem diffuse, which ruins the visual effect. Thus, particle systems are only appropriate for rendering gases.

When using a particle system in a system like ours, particles are simply points in space, flowing freely along the simulated velocity field. Hence, in the context of fluid simulation, particles are often referred to as *mass-less marker particles*.

10.1.1 Updating Particle Positions

The motion of the smoke is already simulated, as described in chapter 8, so the particles should just be animated accordingly. For the use in computer games, a simple low-order integration scheme, such as Euler integration, will often be sufficient. Thus, given the position $\mathbf{p}$ of a particle, the new position $\tilde{\mathbf{p}}$ of the particle can be found by the simple first-order time step

$$\tilde{\mathbf{p}} = \mathbf{p} + \Delta t \ \mathbf{u}(\mathbf{p}), \tag{10.1}$$

where Δt is the time step of the animation and $\mathbf{u}(\mathbf{p})$ is the velocity at the particle position, interpolated from the discrete velocity field.

Problem with Particles when Using GPU

There is a general problem, when using particles for the purpose of this project, since all velocity data is stored in the graphics memory. This means, that the velocity field texture should either be transfered back into client memory, accessible by the CPU, or the updating of the particle positions should be done on the graphics card, i.e., by using a fragment shader.

Transferring the velocity field to client memory is impractical, since it introduces the bottleneck of the graphics bus. The velocity field of a simulation grid of resolution $30 \times 30 \times 30$, using 3D vectors of 32-bit precision, takes up at least 316KB of memory, depending on the internal representation. This amount of data has to be transfered via the bus once per frame, yielding a transfer rate of 9.3MB/s, for simulations running at 30 frames per second. Although this does not sound like much, considering the bandwidth of recent buses, such as 8×AGP, preliminary experiments have shown that this is, in fact, a problem.

Updating the particle positions with a fragment shader is very simple. The particle positions are represented as colors in a texture, and the fragment shader just updates the positions, by adding values, read from the velocity texture. However, this does not solve the problem of transferring data. The particle position texture still needs to be transfered into client memory, in order for the particles to be drawn. The only difference is that the number of particles will normally be a lot lower than the number of cells, depending on the desired effect. Simple rising smoke can be achieved using only a couple of hundred particles.

10.1.2 Rendering Particles

When the particle positions have been updated, the particles can be rendered in numerous ways. Some of the most popular are briefly described in the following.

Dots

The simplest method for rendering particles is to simply draw a dot at each particle position. This method does not produce realistic images of smoke, but can be helpful for testing and for setting up scenes.

Simple Quadrilaterals

In real-time applications, such as games, particles are usually rendered using quadrilateral primitives. By drawing each particle as a quadrilateral, facing the viewer, and mapping the quadrilateral with a texture, the particle can appear volumetric and diffuse. In this way, many kinds of effects can be achieved using only a limited number of particles. This technique is known as bill-boarding, and can for instance be done efficiently by using the OpenGL `GL_ARB_point_sprite` extension [27]. Figure 10.1 shows an example of bill-boarding.

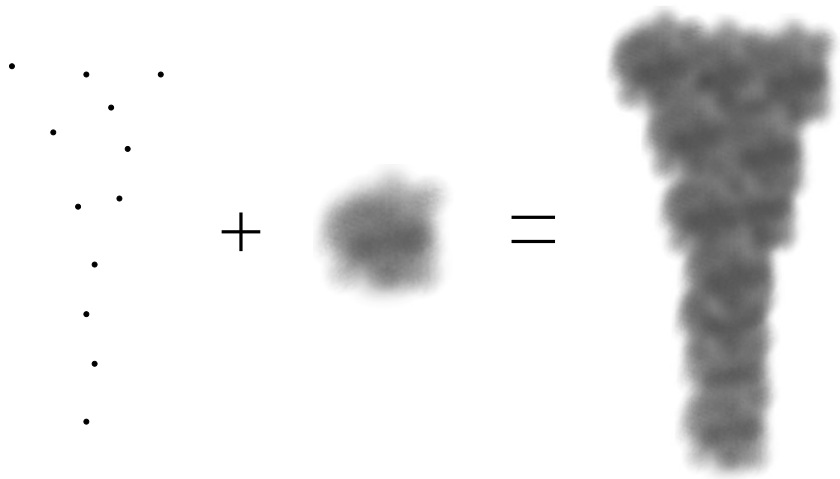


Figure 10.1: An example of bill-boarding. Each particle in the particle system (the dots to the left) is rendered with a texture mapped quadrilateral, facing the viewer (the image in the center), yielding a smoke-like effect (the image to the right).

Advanced Bill-Boarding

The technique of bill-boarding can be taken a step further, by adding rotation, scale, and other operations to the particles, depending on particle properties, such as life time, velocity, etc. This can give a more dynamic effect, since every particle is rendered differently. An advanced example of this is presented in [16].

10.2 Density Texture

In [32], smoke in a 2D fluid simulation is represented using a density field. This density field is simply a scalar field, and is thus represented discretely by a scalar value in the center of each cell in the simulation grid, regardless of the chosen discretization, see chapter 7. The density value of a given cell can be thought of as the amount of smoke in the cell. Empty cells will have the density value 0, and completely full cells will have the density value 1.

As with particle systems, fluids visualized by a density field often seem diffusive, and without a determinable surface. Methods that can generate a polygon mesh from the density field do exist, for instance the *Marching Cubes* algorithm [22]. This method is, however, only applicable in real-time for very small grid resolutions.

Compared to particle methods, the density field method has the advantage that its complexity is not dependent on the amount of fluid, but only on the resolution of discretization of the field. Thus, the simulation will not suddenly run slower, when more fluid is introduced, as is the case with particles. This is an important property when using the simulation in a computer game.

10.2.1 Updating the Density Field

To reflect the behavior of the simulation, the density field needs to be updated, according to the motion of the smoke. In [32], this motion is described by the advective and diffusive terms of the Navier-Stokes equations, equations (3.5) and (3.6). Since the effect of diffusion is already assumed negligible, the motion of the density field can simply be described by

$$\frac{\partial d}{\partial t} = -(\mathbf{u} \cdot \nabla)d, \tag{10.2}$$

where $\mathbf{u}$ is the velocity field and d is the density field. Since this is similar to the advection of the velocity field, it can be solved with the same methods, see section 8.2.

10.2.2 Rendering the Density Field

When using the method described in this thesis, the density field will be represented by a texture, in the same way as all other fields (velocity, pressure, etc.). The visualization of 3D volume data, such as this density field texture, is a complex task, and most of the existing methods are too computationally expensive to be applicable for real-time animations. Recently, however, the power of GPU processing have made high-quality volume rendering possible in interactive-rate applications [24, 23], so real-time rates might be achievable with these methods in the near future.

We will describe a much simpler rendering method, which yields acceptable results in many cases. The method can be seen as a simplification of more advanced techniques, but is in fact merely based on own ad-hoc impressions.

Simple Volume Rendering

Since the density field is represented as a texture, it can be bound as any ordinary texture, and used to texture map primitives. Thus, to display the density field, slices of the density texture can be drawn on layers of quadrilaterals. To give the right impression of transparency, the layers are drawn in order of distance to the eye, starting with the layer farthest away, and each new layer is blended with the previous layers.

Basically, two methods for slicing the texture exist. The quadrilateral primitives can either be placed relative to the camera, as shown in figure 10.2 (a), and the texture indexed using 3D texture coordinates. Or the primitives can be aligned with the density field, as shown in figure 10.2 (b). In this case, three set of layers, parallel to the three base planes in the texture coordinate system, must be drawn and blended with each other, to make the smoke viewable from all directions.

10.3 Hybrids

Because the resolution of the density field will normally be very low, compared to scene details, the density texture method is best suited for large scale effects. Whereas, particle based methods are best suited for rendering small scale effects. This insinuates that the methods could be combined, giving a commonly better visual result. To do this, however, the following issues must first be resolved:

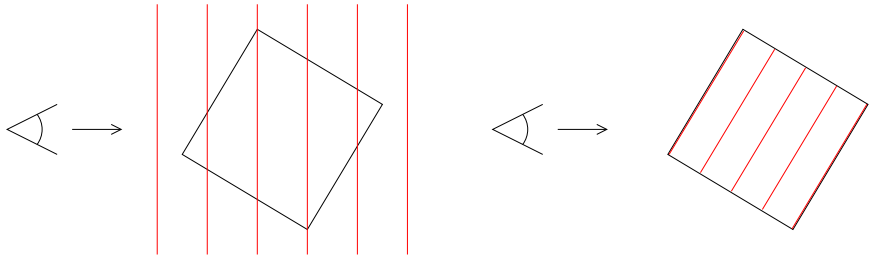


Figure 10.2: Two methods for slicing a volumetric texture. (a) The texture is sliced using planes normal to the viewing direction. (b) The texture is sliced using planes parallel to the three base planes of the texture coordinate system. The red lines indicate the quadrilaterals that slice the volumes.

- When should fluid be represented as particles, and when should it be represented as density?
- If the criteria for either of the representations change, how should the representation be transformed from particles into density and vice versa?

Since the visualization is not the primary focus, to solve these issues is considered out of the scope of this project. We believe, however, that a hybrid model would be the most usable method, for application in a computer game.

Chapter 11

Implementation

As part of this project, some of the methods described has been implemented in a demo application, in order to evaluate their usability in a real-time fluid solver. In this section, the implementation details will be discussed, in order to introduce the reader to some of the problems involved, and to some of the considerations to make, when implementing a real-time fluid solver.

This description will not be a walkthrough of the resulting C, C++, and fragment shader source code, since this would be tedious and probably not very usable. The entire source code for the demo application is available on the CD-ROM accompanying the original of this report, see section 1.4.

11.1 Choosing Platform

A lot of the implementational details depend on the choice of platform. Thus, before discussing the implementation, we need to define what platform we will address.

11.1.1 Graphics API

Through our previous work in the field of computer graphics, we have become familiar with OpenGL on an advanced level. Thus, choosing this API seems obvious. Furthermore, from a quick research in various graphics-related internet fora, the performance in OpenGL is not believed to be significantly lower than in any other APIs, such as DirectX. We will thus base the implementation on the OpenGL API.

One difference between OpenGL and DirectX is the differentiation between core features and extended features. In DirectX all features are core, so a 3D graphics application is dependent on a certain version of the API. In OpenGL the core functionality can be extended through OpenGL extensions [27], which yields that available features depends both on the graphics hardware *and* the drivers. Thus, by choosing the OpenGL API, the choice of graphics hardware becomes more difficult; we need to make sure that the chosen hardware supports all the required extensions.

11.1.2 Graphics Hardware

The market for end-user graphics hardware is dominated by the two manufacturers, ATI and NVIDIA. Thus, the basic choice of graphics hardware stands between these two. We have based our choice of hardware on the following criteria

- **Features:** Obviously, as a minimum the hardware must support the features needed for GPU programming. In recent OpenGL terms this at least requires support of the `GL_ARB_fragment_program` extension [27].
- **Cost:** Since we are on a student budget, the permissible cost of the hardware is very limited. This also covers that the choice of hardware should reflect a generally affordable cost for private users.

With assist from various hardware reviews, our choice has fallen on a discount graphics adapter based on ATI's Radeon 9600 PRO. This card supports the required features at an affordable price[1].

11.1.3 Fragment Shader Language

For the ease of implementation, our first choice of fragment shader programming language, would be Cg [19]. There is already several internet fora related to Cg-programming, which would presumably shorten our learning period. A Cg-compiler and a runtime library for easy handling of Cg-programs is supplied for free by NVIDIA. However, preliminary tests show some incompatibilities between our chosen hardware and the Cg compiler. Thus, Cg is unfortunately not an option. Since the current drivers for the chosen hardware does not support the OpenGL Shading Language, we are forced to the low-level assembly language, as specified by the OpenGL `GL_ARB_fragment_program` extension [27].

11.2 Representing Vector Fields with Texture Maps

Since fragment programs operate on texture memory, while all the fluid simulation steps covered in chapter 8 operate on vector fields and scalar fields, an important part of implementing a fluid solver to run on a GPU is to find an efficient way to represent these vector fields and scalar fields as texture maps.

In the following we will use the term *vector fields* for both vector fields and scalar field, since scalar fields, in this context, are just special cases of vector fields, using only one of the vector components.

11.2.1 3D Texture Maps

Since we are dealing with 3D simulation, and thus 3D vector fields, an obvious choice for field representation would be to use 3D texture maps. In 3D texture maps, pixels are indexed using 3D coordinates (R, S, T), so no abstraction between grid coordinates and texture coordinates is needed.

[1] At the time of purchase, the card was listed at 1300,- DKK (approx. $200 USD).

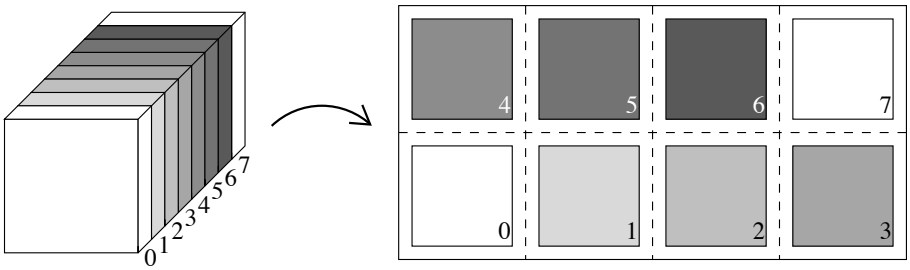


Figure 11.1: The slices of a 3D texture is laid out as tiles on a 2D texture. This figure is the same as figure 5.1, but is presented here for easy reference. The figure is copied from [17].

Unfortunately, when using fragment shaders, we can only output 2D memory arrays. Thus, to update a vector field represented by a 3D texture map, each slice in the 3D texture map has to be updated separately. This requires one *render pass* per slice, which is inefficient.

11.2.2 Flat 3D Texture Maps

To come by the inconvenience of 3D texture maps, Harris et al. use specially formatted 2D texture maps, which they call *flat 3D textures* [17]. In a flat 3D texture, each slice in the corresponding 3D texture map is represented by a rectangular tile in a 2D texture map, as shown in figure 11.1. Every slice can be rendered independently by indexing the texture correctly, and since the entire field is held in a 2D texture map, all slices can be updated in one render pass.

Using flat 3D textures, thus, require some mapping between grid coordinates and texture coordinates. In [17], this mapping is done inside the fragment shaders, by introducing a precomputed 1D offset texture. At position k, this texture holds the absolute coordinates $\texttt{offset}_{1D}(k) = (x_k, y_k)$ of the lower left pixel of tile k, see figure 11.2. The fragment shaders are executed with 3D texture coordinates, and the absolute 2D texture coordinate for indexing cell (i, j, k) in the flat 3D texture can then be found in the fragment shader by

$$(x, y)_{i,j,k} = (i, j) + \texttt{offset}_{1D}(k) = (i, j) + (x_k, y_k), \quad c_{i,j,k} \in \mathcal{G}. \tag{11.1}$$

This method is preferable, since the grid is indexed with 3D coordinates, which is intuitive. However, this lookup requires a texture read-instruction per rendered pixel, which is inefficient. Furthermore, this method is complicated by the chosen hardware, since it is unable to use integer texture coordinates; depending on the texture setup, texture coordinates are either clamped to the range $[0, 1]$ or the integer part is discarded, leaving only the fractional part (also in the range $[0, 1]$).

We solve this in the following way. Usually only the offsets to the nearest two tiles are needed, and since the positions of the tiles in the flat 3D texture are constant, the relative offsets to neighboring tiles can be precomputed and passed to the fragment shader as two separate texture coordinates, thus, saving a texture lookup of the offset per pixel.

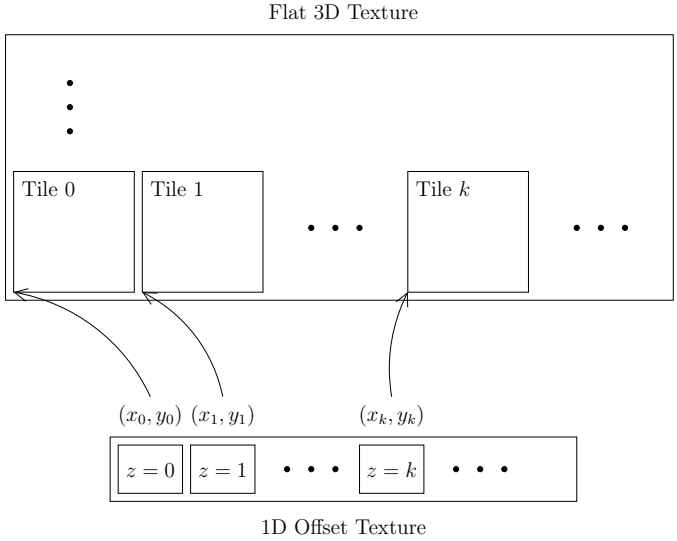


Figure 11.2: The z coordinate is looked up in a 1D texture, to get the (x, y)-offset to the corresponding tile in the flat 3D texture.

Unfortunately, the semi-Lagrangian advection scheme, described in section 8.2.2, requires access to other than just the neighboring tiles, since a point can be advected more than one cell in the z direction. Thus, a means of looking up offsets from a texture map is inevitable.

We solve this by introducing a 2D offset table. At position (k_1, k_2) in this table we store the relative offset, needed to jump from tile k_1 to tile k_2. I.e.,

$$(x, y)_{i,j,k_2} = (x, y)_{i,j,k_1} + \texttt{offset}_{2\text{D}}(k_1, k_2), \quad c_{i,j,k_1}, c_{i,j,k_2} \in \mathcal{G}, \tag{11.2}$$

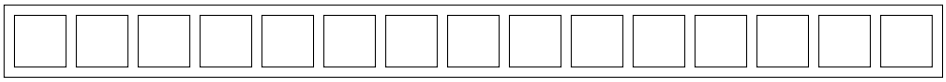
where $(x, y)_{i,j,k}$ is the absolute 2D texture coordinates of the cell $c_{i,j,k} \in \mathcal{G}$ in the flat 3D texture. When a fragment shader is executed on a tile k, the offsets $\texttt{offset}_{2\text{D}}(k, k-1)$ and $\texttt{offset}_{2\text{D}}(k, k+1)$ are issued to the shader as texture coordinates. For the advection shader, the 2D offset table is supplied as a 2D texture map, so the relative offsets can be looked up dynamically, as well. For consistency, the relative offset from a tile k to it self is set to

$$\texttt{offset}_{2\text{D}}(k, k) = (0, 0). \tag{11.3}$$

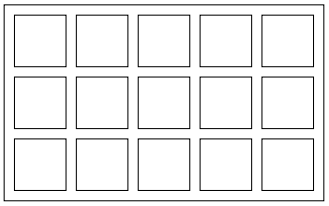
Flat 3D Texture Layout

A thing that should be considered, when using flat 3D textures, is how the tiles are laid out across the 2D texture map. Figure 11.3 shows three different layouts of a field of dimensions $15 \times 15 \times 15$.

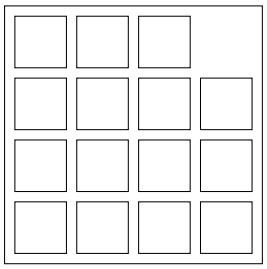
For easy indexing and offsetting, having all tiles in one row seems preferable, as in figure 11.3 (a). However, in OpenGL the width and height of 2D texture maps are restricted



(a)



(b)



(c)

Figure 11.3: Different possibilities for laying out a $15 \times 15 \times 15$ grid in a flat 3D texture. (a) In the simplest form, the tiles are laid in one row. (b) A natural way is to use the divisors of the depth, in this case 3 and 4. (c) The area covered by the tiles does not have to be a square. Each of the small squares represent a tile of 15×15 pixels.

to powers of 2 ($2^n, n \in \mathbb{N}$)[2], and they are upwards bound by a driver-dependent limit. Thus, laying all tiles in one row could easily conflict with that limit. For instance, a field with a resolution of $50 \times 50 \times 50$ would require a width of at least $50 \cdot 50 = 2500$. Since the width must be a power of 2, a texture map of width 4096 is needed. Also, since a texture map height of 64 is required, the total amount of pixels in the texture map is $4096 \cdot 64 = 262144$ pixels. Since a vector is represented by pixels of three 32-bit words, this yields a total amount of memory allocated for the texture map of 3.0 megabytes. But only $50 \cdot 50 \cdot 50 = 125000$ pixels ≈ 1.4 megabytes are actually used.

This leads to the idea that the tiles should be laid out differently. A natural way to try to lower the memory usage would be to use divisors – if they exist – of the number of tiles. This is demonstrated in figure 11.3 (b). If the tiles in the $50 \times 50 \times 50$ grid from before are laid out as 10 tiles by 5 tiles, only $512 \cdot 256 = 131072$ pixels = 2.0 megabytes is required. However, this is not always the best way to lay out tiles either, and what should happen if no divisors are found?

Actually, what we want is to find the minimum amount of memory required to lay out a number of tiles. Finding this minimum is not a complex task; all possibilities can be checked in time $\mathcal{O}(n)$, where n is the number of tiles. However, we will just use the following simple method.

First, the width of the 2D texture map is computed as

$$texture\ width = 2^n, \qquad 2^n > \left\lceil \sqrt{grid\ depth} \right\rceil \cdot grid\ width > 2^{n-1}. \tag{11.4}$$

Second, the number of tiles in the x direction is computed as

$$tiles\ in\ x = \left\lfloor \frac{texture\ width}{grid\ width} \right\rfloor. \tag{11.5}$$

The number of tiles in the y direction can now be computed as

$$tiles\ in\ y = \left\lceil \frac{grid\ depth}{tiles\ in\ x} \right\rceil. \tag{11.6}$$

Finally, the height of the 2D texture map is computed as

$$texture\ height = 2^m, \qquad 2^m > tiles\ in\ y \cdot grid\ height > 2^{m-1}. \tag{11.7}$$

We note that this method is not optimal. However, in many cases it will, in fact, find an optimal solution.

11.2.3 Representing a Vector With a Pixel

The elements of the texture map, pixels, are used to represent elements in the discretized vector fields, vectors. This is done simply by letting the R, G, and B values of a pixel represent the x, y, and z components of the corresponding vector.

[2]This is based on the chosen hardware. Some other graphics adapters support the OpenGL extension `GL_ARB_texture_non_power_of_two`, which allows arbitrary texture map sizes.

When using regular OpenGL texture maps, values are represented with fixed-point precision, and are clamped to the interval $[0, 1]$. To be able to represent values outside this interval, simple linear projections can be used to scale and offset values when read from and written to a texture map. This has the following two drawbacks:

- The projections require extra fragment shader instructions, when reading and writing values in texture maps.
- Projections are subject to numerical dissipation, since values are scaled up and down successively.

Most modern graphics cards support the use of 32-bit single precision IEEE floating point values in texture maps, which makes the mentioned projection unnecessary. However, in preliminary tests, slower processing of 32-bit textures have been experienced. Thus, we will keep the projection available, so both precisions can be used.

11.3 Implementing the Solver

In this section we will discuss the implementation of our solver into fragment shaders for execution on a GPU. We will go through each step of the solver, as explained in chapter 8, and explain the implementational details of the resulting fragment shaders. The fragment shader names refer to the source code files in the `/shaders/` directory on the accompanying CD-ROM.

11.3.1 Advection

Since the purpose of our solver is real-time simulation, stability is crucial. Thus, we will implement advection using the semi-Lagrangian scheme, as described in section 8.2.2. This can be implemented in a single fragment shader, defined in `advection.fp`.

The `advection.fp` fragment shader is implemented in such a way, that the same shader can be used for advecting the velocity texture, the temperature texture, and the density texture.

11.3.2 Thermal Buoyancy

The thermal buoyancy step involves two fragment shaders. First, the temperature texture is advected using the `advection.fp` shader. The buoyancy force is then computed by equation (8.37) and added to the external force texture. This is done by the fragment shader defined in `thermal_buoyancy.fp`.

For simplicity, the temperature is not represented as an absolute temperature. Since computing the buoyancy force only requires the temperature relative to the mean temperature, we can represent the relative temperature directly. Hence, we save the representation of a mean temperature, as well as the per-pixel computation of the relative temperature as $T_{relative} = T_{absolute} - T_{mean}$.

11.3.3 Vorticity Confinement

For vorticity confinement we will need two fragment shaders. In the first shader, `vorticity.fp`, we compute the vorticity vectors $\boldsymbol{\omega}$, given by equation (8.35), and store them in the vorticity texture. For efficiency, this fragment shader also computes the length $|\boldsymbol{\omega}|$, and stores it in the alpha component of the vorticity texture.

In the second fragment shader, `vorticity_confinement.fp`, we first compute the $\mathbf{N}$ vectors, given by equation (8.33) – this is where the lengths of the vorticity vectors come in handy. Then we compute the confinement force, and add it to the external force texture.

11.3.4 External Forces

Since all external forces are added to the external force texture, adding these into the velocity field requires only one fragment shader, `add_forces.fp`. This shader simply scales the external force with the time-step, and adds it to the velocity texture.

11.3.5 Mass-Conservation

To remove divergence in the velocity texture, we need three fragment shaders. In the first shader, `divergence.fp`, we compute the divergence D of the velocity texture, given by equation (8.24), and store it in the divergence texture. For efficiency, we actually compute hD in order to reduce the next fragment shader as much as possible.

The fragment shader `pressure_solve.fp` implements a single step of the Jacobi solver. The Jacobi-step is computed as given by equation (8.28), and the result is stored in the pressure texture. The approximated pressure solution is obtained by running the fragment shader repeatedly, using the recently updated pressure texture as input – see chapter 12 for testing with different number of iterations.

Finally, when the pressure is computed, the fragment shader `pressure_adjust.fp` calculates the pressure-gradient, and subtracts it from the velocity texture, according to equation (8.29).

11.3.6 Upholding Boundary Conditions

Even though boundary conditions are very similar for the boundaries of the fluid volume and the internal objects, we will uphold these boundary conditions in two different ways.

11.3.7 Volume Boundaries

Since the boundaries of the fluid volume are always defined in the same way, the handling of these boundaries are simpler, than the handling of the internal boundaries. We update the volume boundaries using six different fragment shaders, which are executed on the left and right columns of pixels in each tile, the lower and upper rows of pixels in each tile, and on the first and last tiles in the grid, respectively. Each pixel is updated from the adjacent interior pixel. The update of the six boundaries are shown in figure 11.4.

For this purpose we have made six separate fragment shaders, `left_boundary.fp`, `right_boundary.fp`, `lower_boundary.fp`, `upper_boundary.fp`, `front_boundary.fp`, and `back_boundary.fp`. To enable easy experimenting with different boundary conditions, we

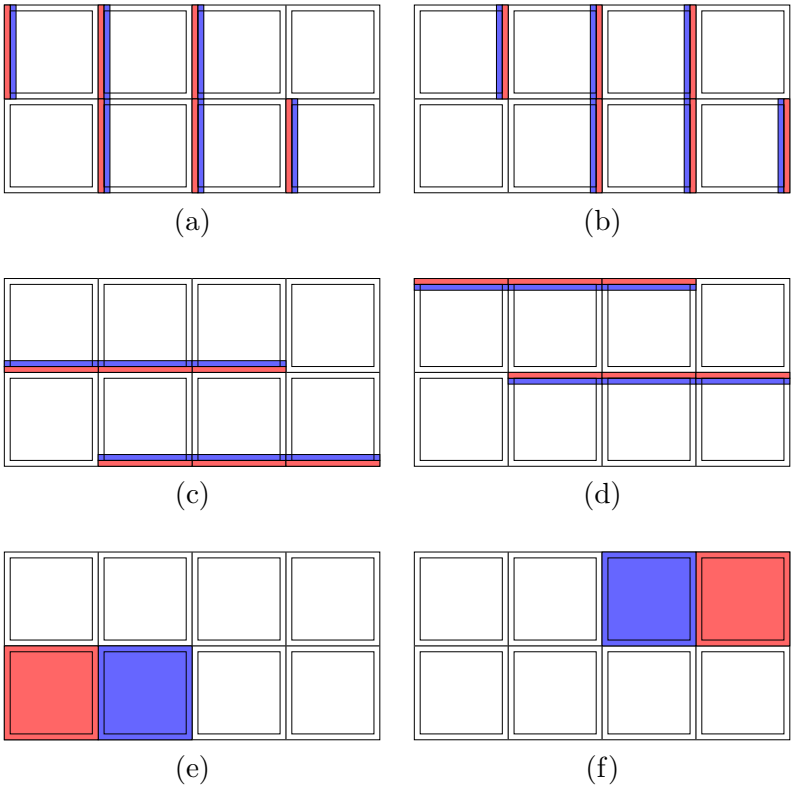


Figure 11.4: The boundary volume is updated in six steps. Blue areas are read and red areas are written. First the left column (a) and right column (b) of pixels in each tile are updated from their neighbor columns. Then the lower row (c) and upper row (d) of pixels in each tile are updated from their neighbor rows. Finally, the front tile (e) and back tile (f) are updated from their neighbor tiles.

call each fragment shader with a boundary dependent vector $\mathbf{a}$. Pixel values $\mathbf{b}$ on the boundaries are computed as

$$\begin{aligned}
\mathbf{b}_{0,j,k} &= \mathbf{a}_{left} * \mathbf{b}_{1,j,k}, \\
\mathbf{b}_{W-1,j,k} &= \mathbf{a}_{right} * \mathbf{b}_{W-2,j,k}, \\
\mathbf{b}_{i,0,k} &= \mathbf{a}_{lower} * \mathbf{b}_{i,1,k}, \\
\mathbf{b}_{i,H-1,k} &= \mathbf{a}_{upper} * \mathbf{b}_{i,H-2,k}, \\
\mathbf{b}_{i,j,0} &= \mathbf{a}_{front} * \mathbf{b}_{i,j,1}, \\
\mathbf{b}_{i,j,D-1} &= \mathbf{a}_{back} * \mathbf{b}_{i,j,D-2},
\end{aligned} \tag{11.8}$$

where $c_{i,j,k} \in \mathcal{G},\ 0 \leq i < W,\ 0 \leq j < H,\ 0 \leq k < D$, and $*$ denote coordinatewise multiplication. For instance, to set the left boundary as a free-slip wall, we would set left boundary in the velocity field as

$$\mathbf{a}_{left} = (-1, 1, 1)^T, \tag{11.9}$$

yielding

$$\mathbf{b}_{0,j,k} = (-b^x_{1,j,k}, b^y_{1,j,k}, b^z_{1,j,k})^T. \tag{11.10}$$

In this way, we can use the same approach for setting boundary condition in other fields. By setting $\mathbf{a} = (1, 1, 1)^T$, the boundary values are simply copies of their interior neighbor values, as wanted in for instance the density field. To save simulation time, boundary conditions are not enforced on fields, where the boundary is never used, for instance in our accumulating force field.

11.3.8 Internal Boundaries

Updating boundaries in the interior of the fluid volume is a bit more difficult. Since we are interested in handling arbitrary internal boundaries, we cannot easily tell, whether a cell in the grid should be updated as a left, right, lower, upper, front, or back boundary, or maybe even several or all of these.

In [21], this has been done for a staggered representation, by representing obstacles with an *obstacle map*. This obstacle map is ordered in the same way as all other fields. Since a staggered representation is used, obstacles can be defined on face-level. This is done by representing obstacle faces with 0 in the obstacle map, and representing faces where fluid can flow freely with 1. Boundary conditions can then be set, by masking out values with the obstacle map, i.e. velocity values are multiplied with the obstacle map value, to yield either 0 (if the face is a boundary) or the original velocity (if the face is not a boundary). This method is very efficient, since only a single render pass with a simple fragment shader is needed for upholding boundary conditions.

In this context, however, we use a collocated representation, so face velocities cannot be explicitly set to 0. By adopting the obstacle map for representing objects, the upholding of boundary conditions can be incorporated into the solver steps. In this way, every fetch of a neighbor value requires a lookup of the obstacle map value for the according neighbor cell, thus, doubling the number of texture-reads. The result of the solver step should then take account of the obstacle map value, to respect any boundaries, thus, it requires extra

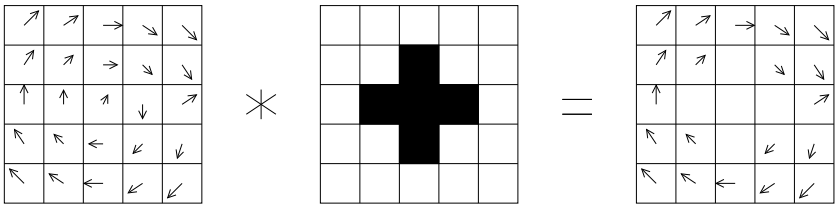


Figure 11.5: The velocity field to the left is masked with the obstacle map in the center, producing the velocity field to the right. In the obstacle map, white cells represent fluid, and black cells represent obstacles. In the resulting velocity field, velocities become **0** in cells occupied by obstacles.

arithmetics to correctly handle the boundary. Since this is a very time-expensive solution, it does not fit very well with our hope for real-time animation.

For simplicity, we do adopt the obstacle map, for representing internal objects. As described in section 9.2, a possible way to handle boundary conditions, is to set velocities to 0 in obstacle cells. Although it is uncertain how well this method works, we will try this approach, and see how it goes. This should be evaluated in the tests in chapter 12.

By choosing this approach, the internal boundaries can be handled by the simple fragment shader `mask_field.fp`. Every time the velocity field has been updated, this fragment shader simply masks out velocity values in obstacle cells. Boundaries are thus respected, without changing any of the other fragment shaders. The effect of the `mask_field.fp` shader is illustrated in figure 11.5.

Boundaries in other fields than the velocity field are handled similarly, however, it is not necessary to set boundary conditions on all fields. To save simulation time, we only enforce boundary conditions on the relevant fields.

11.4 Visualization

To present the simulations produced by our solver visually, a density field is advected along with fluid velocity, as described in section 10.2. Since this density field is also represented by a texture map, we can take advantage of all four color components, to represent colored smoke and transparent smoke. By this, colors and transparencies can be specified per cell, which allows us to handle multiple colors of smoke in the same density field, and because of the generality of the advection shader, the colors are automatically blended, when smoke of different colors interact. This will be demonstrated in chapter 12.

To render the density field, tiles are drawn in order of distance to the viewer, starting with the tiles farthest away. The density field is laid out as tiles in a 2D texture, each tile representing a slice of the volumetric density field at a given z coordinate. Thus, we cannot obtain a fully volumetric rendering of the density field, just by rendering each tile. If the viewing direction is parallel to the $x - y$ plane, the tiles will be rendered as thin plates, and the volumetric feeling is gone.

Obviously, this should be fixed if the solver was used in a full animation system. However, as stated in chapter 6, the visualization is purely ment to be a means of evaluating our

solver. Thus, we let this stand – not as an error, but as an obvious point of improvement. As will be shown in chapter 12, where the solver is put to some tests, the visualization produce nice images of smoke, when viewed from the right direction.

11.5 Optimizations

During the work with the solver, we have come to think of some ways to optimize some of the processes. In this section we mention the ones we have not had time to implement.

11.5.1 Memory Usage

For simplicity, we use the same type of OpenGL texture format for all the discretized fields. This means that the textures used to represent the temperature, divergence, and pressure fields allocate all four color components. This is taking up a lot of graphics memory, which could be freed very easily, e.g., by using a texture format with only one channel, such as the `GL_INTENSITY` format. We presume that lowering the memory use will increase the processing speed, since less memory will have to be read and written (only one component per read/write, compared to four with the current texture format).

11.5.2 Pressure Solver

Intuitively, the mass-conservation step is the most expensive part of the solver, since the pressure solving step is executed far more times than the other fragment shaders. This is thus an obvious place to look for optimization possibilities.

Apart from lowering the memory use, as mentioned above, one idea is to represent divergence and pressure in the same texture. Both the divergence and the pressure of a given cell can then be read in a single texture-read, thus, saving one texture-read per cell per iteration.

11.5.3 Pixel Packing

Representing scalar fields with regular texture maps, with 3 or 4 components (RGB and $RGBA$, respectively), is obviously subject to an overhead in memory usage. And even though scalar fields can be represented with single-channel texture maps, we still use one texture-read operation per scalar value.

In [17] scalar values of the pressure field are *packed*, as shown in figure 11.6. This packing not only reduces the memory usage, it also improves the performance of the iterative Poisson-solver. First of all, less texture-read operations are required, since a single texture-read operation actually reads four scalar values. Second, the convergence of the iterative solver can be improved by using the Red-Black Gauss-Seidel variation of the Jacobi method [9, 17].

Some of the steps in our solver require the use of all three velocity components in a single computation (for instance when computing the divergence, see section 8.4). In these cases, we would need three texture-reads to get the three components from separate scalar fields. But we would still read four values with every texture read, thus, an average of 3/4

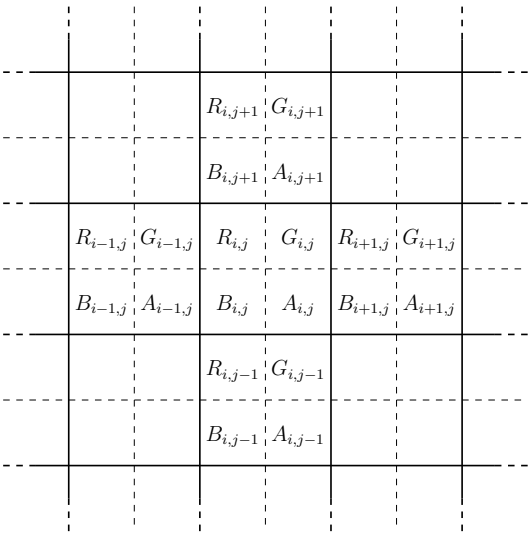


Figure 11.6: Pixel packing. A scalar field represented with an $RGBA$ texture map. The values $R_{i,j}$, $G_{i,j}$, $B_{i,j}$, and $A_{i,j}$ are the values of the four components of pixel (i, j). The bold lines indicate the pixels of the texture map. The dashed lines indicate the virtual splitting into discrete scalar field positions.

texture-reads per vector when using pixel packing, compared to 1 texture-read per vector, when using the vector-based approach.

In [32], a 2D velocity field is represented by two separate scalar arrays. The steps of advection and diffusion are performed on the scalar arrays separately, so in these steps only updates of scalar fields are made. Also, when enforcing mass-conservation (see section 8.4) the velocity components are updated separately. This indicates, that representing our vector fields as three packed scalar fields is plausible.

To give an idea of the savings obtainable, when using packed scalar fields instead of regular vector fields, table 11.1 gives a comparison of the amount of texture-reads per processed value for selected fragment shaders. The table shows a significant saving in number of texture-read operations, when using pixel packing. Thus, we believe that using pixel packing would speed up the solver.

Fragment shader	Instructions per cell without pixel packing	Instructions per cell with pixel packing
`add_source.fp`	$\frac{1\ target+1\ source}{1\ target} = 2$	$3 \cdot \frac{1\ target+1\ source}{4\ targets} = 3/2$
`vorticity.fp`	$\frac{6\ velocities}{1\ vorticity} = 6$	$3 \cdot \frac{4\ velocities}{4\ vorticities} = 3$
`vorticity_confinement.fp`	$\frac{1\ force+7\ vorticities}{1\ force} = 8$	$3 \cdot \frac{1\ force+6\ vorticities}{4\ velocities} = 21/4$
`divergence.fp`	$\frac{6\ velocities}{1\ divergence} = 6$	$\frac{9\ velocities}{4\ divergences} = 9/4$
`pressure_solve.fp`	$\frac{1\ divergence+6\ pressures}{1\ pressure} = 7$	$\frac{1\ divergence+6\ pressures}{4\ pressures} = 7/4$
`velocity_adjust.fp`	$\frac{6\ pressures+1\ velocity}{1\ velocity} = 7$	$3 \cdot \frac{3\ pressures+1\ velocity}{4\ velocities} = 3$

Table 11.1: Comparison of the number of texture-read operations per processed value, in selected fragment shaders, with and without using pixel packing. This table shows that a significant reduction in number of texture-read operations can be obtained when using pixel packing.

Chapter 12

Results

In this chapter, we will put our solver to some tests, to evaluate the methods described in previous chapters. To do this, section 12.1 first sums up the criteria, which we will use to evaluate the solver. Then, in section 12.2, we will present some test scenarios and discuss the performance of the solver in each test. Finally, in section 12.3, we will sum up the tests, with an overall evaluation of the solver.

12.1 Test Criteria

In order to construct scenarios for testing the solver, we need to make clear what aspects we wish to test. The primary criteria we have proposed for our solver sum up to the following:

- **Smoke-Like Behavior:** Our main objectives is to simulate smoke. So an obvious criteria is: Does the simulated fluid appear smoke-like? And as part of this, do the individual steps add the expected effects?

- **Solver Speed:** We have laid a great deal of importance on the speed of the solver. Can the solver actually run in real-time, and what frame rates are actually possible?

- **Interaction With Objects:** We have implemented a very simple way of simulating fluid/object interaction. Does the simulated fluid behave convincingly around inserted objects?

12.1.1 Additional Experiments

As an extension to the primary criteria mentioned above, there are some implementational aspects of the solver, we wish to test. These aspects are mostly implementational details, such as the choice of texture map precision, and the number of iterations used, when solving pressure. We will test these aspects to some extend. Not to determine whether things work or not, but to clarify their capability and explore their consequences.

12.1.2 Things We Will Not Test

The overall performance of the solver depends on a lot of basic operations, such as the handling and indexing of flat 3D textures, and the handling and execution of fragment shaders. To test all these operations explicitly would be an enormous task, and is considered unnecessary. Many of the basic operations are so important for the overall performance of the solver, that if any of them fail the solver will clearly not work. We will thus assume, that if the solver generally appear to work, on a level that enables us to evaluate the above criteria, the basic operations also work properly. Hence, we will not explicitly test all basic operations.

As stated in chapter 6, we have not put a great effort into visualizing our animations. However, since many of the aspects of the solver are evaluated based on their visual appearance, an evaluation of the visualization to some low extend, would be in place. We will not explicitly test the visualization, but simply present some pros and cons of the chosen visualization method.

12.2 Test Scenarios

In this section, we will describe the scenarios we have chosen to use for evaluating the solver. For each scenario, we will show the results from running the solver with the given scenario, and discuss any interesting issues in the result. Most of the tests are output to animations in AVI-format. Since the results are better understood by looking at the actual animations, the animations of all tests can be found on the CD-ROM accompanying this report, by the filenames `/test/<testname>.avi`. To present the test results in the report, we show series of snapshots from these animations. Since images are better viewed on a computer screen, these images can also be found in the JPEG format on the CD-ROM, in the files `/test/<testname>/<testname>_<framenumber>.jpg`. Test 1a is a little different than the other tests. There is no animation file for this test, and the images are named differently. Their name will be mentioned in their context.

12.2.1 Test 1: Iterative Pressure Solver

The pressure solver is an important part of the solver. We thus want to determine the precision as the first test, since any misbehavior in the pressure is expected to show up in later tests.

Test 1a: Divergence and Pressure

First we need to determine if the divergence and pressure fields are even computed correctly. To do this, we set up a velocity field with six velocity vectors, pointing towards each other. A tile from the velocity field texture is shown in figure 12.1 (a). The divergence of this velocity field should be high in the cell between the velocities, and there should be a cell with low divergence on the opposite side of each cell. The corresponding tile of the divergence field is shown in figure 12.1 (b). The result is what we expected.

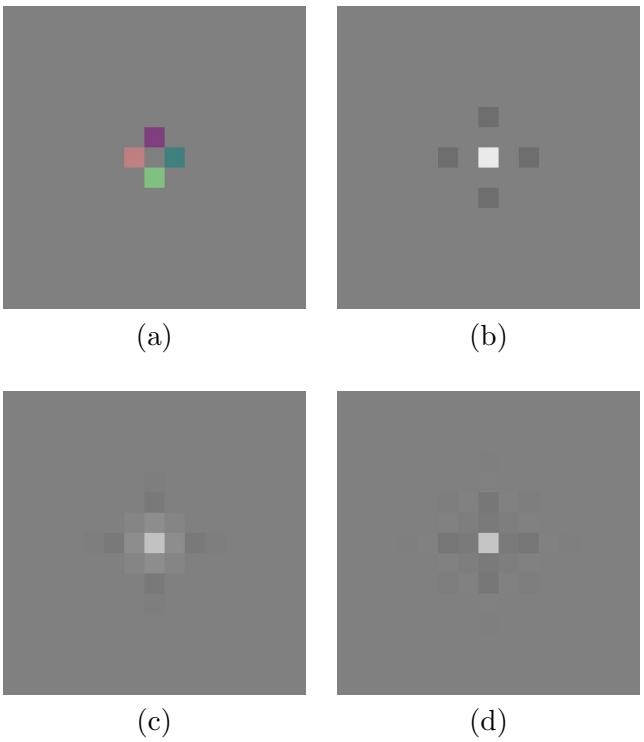
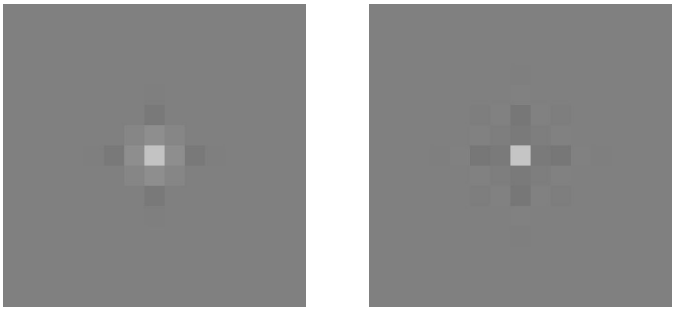


Figure 12.1: Images from one iteration with the pressure solver. These images are named `/test/test1a/fig1_a.jpg`, `/test/test1a/fig1_b.jpg`, `/test/test1a/fig1_c.jpg`, and `/test/test1a/fig1_d.jpg` on the CD-ROM, respectively. The velocity field texture (a) is setup so velocities point towards each other, with a single cell in between. The red component represents the x component of the velocity, and the green component represents the y component. Gray represents 0. The resulting divergence field texture (b) has high divergence (white area) in the center cell and low divergence (dark areas) on the opposite side of the velocities. The pressure field (c) is computed from the divergence field using 20 Jacobi iterations. The high divergence result in high pressure (light gray), the low divergence result in low pressure (dark gray), and the pressure field is generally smoothed. The gradient of the pressure field is subtracted from the velocity field, and the divergence is recomputed (d). The divergence field should now be 0, but there is still some divergence left.



(a) (b)

Figure 12.2: Images from running the pressure solver with 100 iterations. These images are named `/test/test1a/fig2_a.jpg` and `/test/test1a/fig2_a.jpg` on the CD-ROM. (a) The pressure field. (b) The divergence of the velocity field after subtracting the pressure gradient.

The pressure field is computed, and the result, shown in figure 12.1 (c), is what we expected. The high divergence in figure 12.1 (b) results in high pressure, and low divergence results in low pressure.

The pressure gradient is then computed and subtracted from the velocity field. The divergence of this updated velocity field is shown in figure 12.1 (d). To our suprise, the divergence is not 0; it *is* a little closer to 0, but it is not 0! By repeating the whole mass-conservation step, the divergence gets smaller and smaller, but after 5 steps, there is still some divergence left.

In an attempt to explain this misbehavior, we run the pressure solver on the same initial velocity field, but with different number of iterations. Figure 12.2 shows the pressure field when using 100 iterations, and the divergence velocity field after subtracting the pressure gradient. The resemblence with figure 12.1 (c) and (d) is remarkable. The increased number of iterations, it seems, does not result in any significant increase in precision.

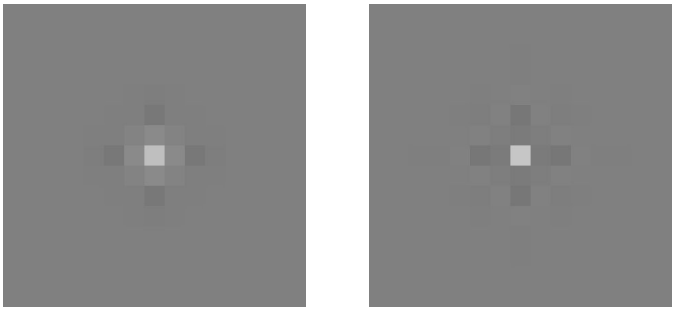
This leads us to try the pressure solver with a far lower number of iterations. Figure 12.3 shows the results. The lack of difference from figure 12.1 and 12.2 is striking.

Having this experience in mind, we state the following:

- **Number of Iterations:** It seems that increasing the number of iterations above 20 does not have a significant effect on the precision of the solver. Thus, we will run the rest of the test using 20 iterations. One of the tests, however, should show simulations with different numbers of interations, in order to analyze the effect on the quality of the simulations, as well as on the frame rates.

- **Conservation of Mass:** Since the mass-conservation step does not remove divergence in a single step, we will expect the test simulations to suffer from some volume-loss. It is not certain to what extend, but it should be kept in mind.

We will now move on to the rest of the tests.



(a) (b)

Figure 12.3: Images from running the pressure solver with 5 iterations. These images are named `/test/test1a/fig3_a.jpg` and `/test/test1a/fig3_a.jpg` on the CD-ROM. (a) The pressure field. (b) The divergence of the velocity field after subtracting the pressure gradient.

12.2.2 Test 2: Advection

Advection is one of the most important features of fluid motion. Thus, we should make it clear, that when put into motion, e.g. by a vent force, the fluid advects accordingly.

Test 2a: Advection of the Velocity Field

To test the effect of the fluid advecting it self, we set up a scene as shown in figure 12.4 on page 86. We run the test, enabling only the velocity source, velocity advection and mass-conservation steps. To get a clearer view of the advection, we display the velocity vector field. We only display vectors with a magnitude greater than some minimum, to avoid a lot of small vectors disturbing the picture. Vectors are colored according to their direction and magnitude: The x component is colored red, y is colored green, and z is colored blue.

When running this scenario, velocities should advect from the faces of the simulation box towards the center. The six streams should arrive at the center simultaneously, and be forced outwards in diagonal stream due to mass-conservation.

Figure 12.5 on page 87 shows images from running test 2a. We see that velocities do in fact advect towards the center of the simulation box, and the six streams reach the center simultaneously.

When the streams touch each other, the incompressibility forces the fluid outwards in diagonal streams as expected. We see, however, that these outgoing streams are not perfectly symetric. We have searched the advection fragment shader thoroughly for errors related to this, but have come up with nothing. Our presumptions are, thus, that this is caused by a slight imprecision in the indexing of the textures, which may even be amplified by the use of interpolation.

We cannot conclude that advection of the velocity field is working as supposed, but overall the velocity is advected almost as it should. However, because of the small size of this problem, it is not expected to have any visual consequences in the rest of the tests.

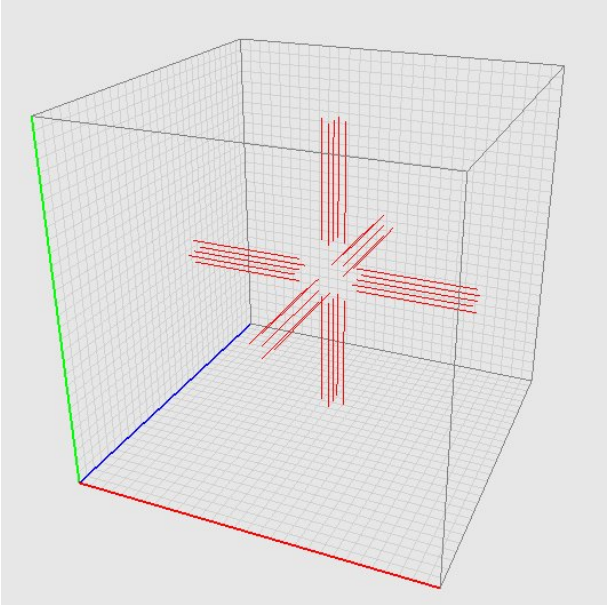


Figure 12.4: Setup for test 2a. Six vent positioned on the faces of the simulation box, pointing towards the center of the box. The forces are indicated by red lines.

Test 2b: Advection of the Density Field

This test tests the ability of another field to advect freely along the velocity field. For this test we use the setup shown in figure 12.6 on page 88. This test is run with the following steps enabled: velocity source, velocity advection, density source, and density advection. The result should be a rising stream of smoke, that hit the ceiling, and flow outwards and down the sides of the simulation box.

Images from this simulation can be seen in figure 12.7 on page 89. We see that the density is in fact advected upwards. When reaching the cieling the density does in fact spread out and flow down the walls of the simulation box.

Test 2c: Advection of a Thin Stream of Density

In [12], the semi-Lagrangian advection scheme is claimed to suffer from severe volume-loss. Although this should also come up in later tests, it is demonstrated here, in order to relate it to the advection scheme. For this purpose we use the setup in figure 12.8 on page 90, and enable only the velocity and density source and advection steps. The venting force is kept at the same power and direction as in the previous test, and we thus expect similar behavior.

From the images in figure 12.9 on page 91 it can be seen that the smoke stream does not evolve as quickly as in test 2b. Further more, when hitting the cieling, the smoke does not spread out in the same degree. Although some of this behavior may be annotatable to other aspects, such as the visual presentation of the smoke, it seems that the advection do suffer from volume-loss. To decrease the infliction of this volume-loss on the rest of the

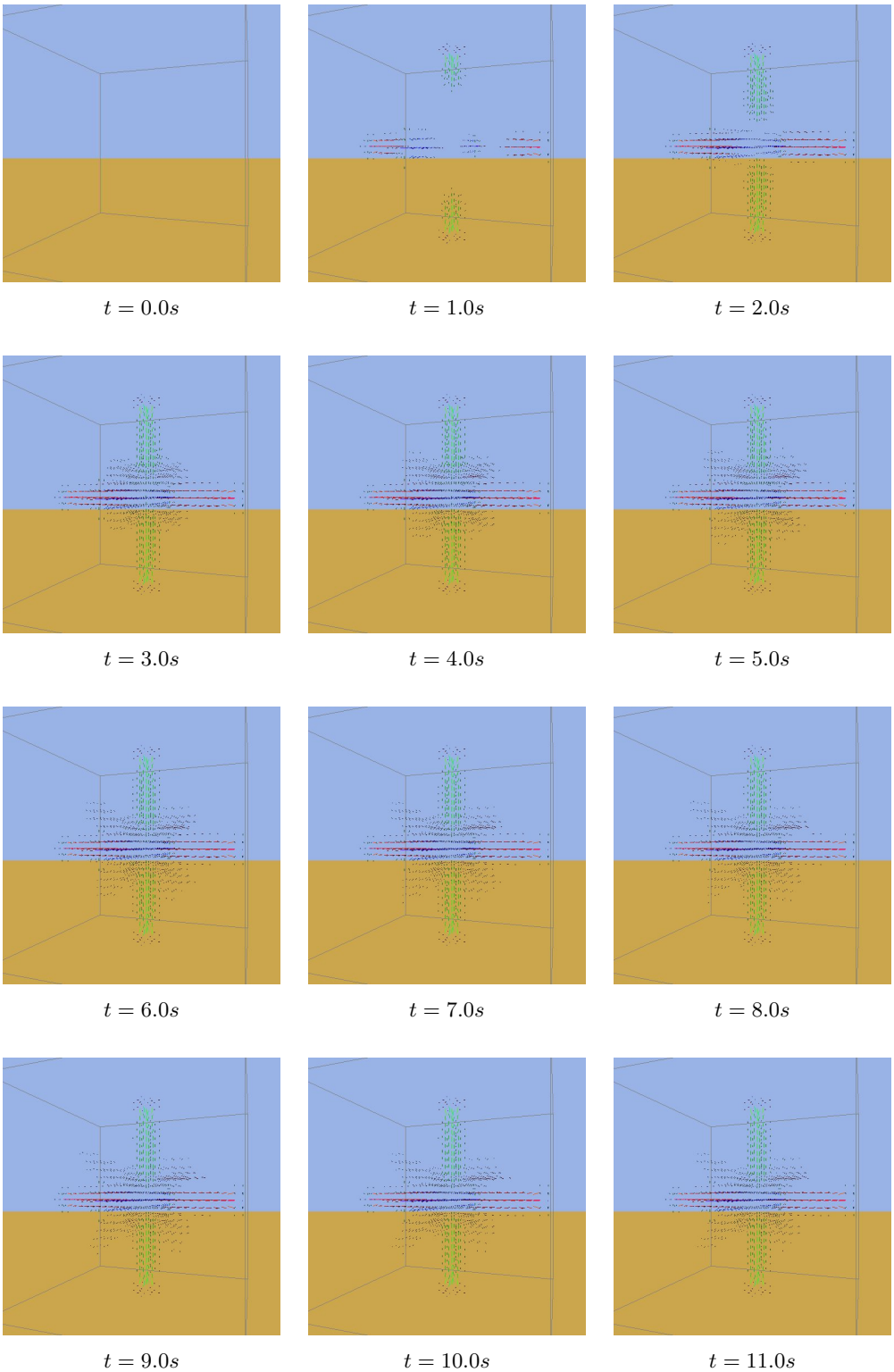


Figure 12.5: Images from test 2a.

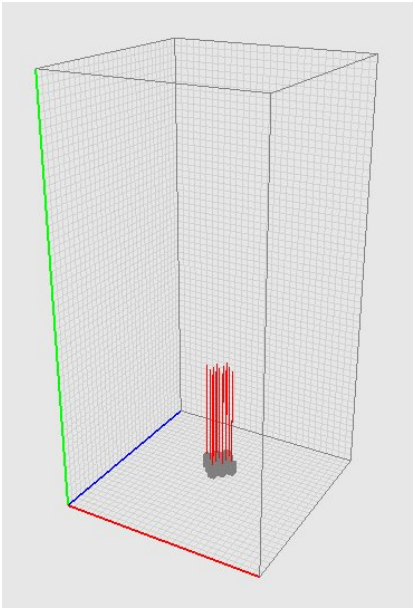


Figure 12.6: Setup for test 2b. A vent force as well as a density source is inserted in the bottom of the simulation box. Again, forces are indicated by red lines, and the gray boxes indicates cell with density sources.

tests we will use wide density sources, such as the one in test 2b, rather than thin sources as in this test.

We notice that the smoke stream is actually not advecting straight upwards, but is moving a little in the possitive directions of x and z. We impute this on the imprecision of the advection step, noted in test 2a.

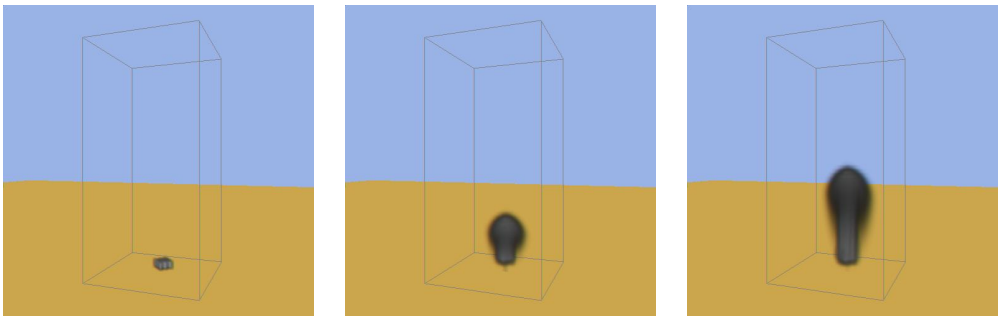
12.2.3 Test 3: Thermal Buoyancy

Differences in temperature should give rise to according forces: Heat should cause smoke to rise, and cold should cause smoke to fall down.

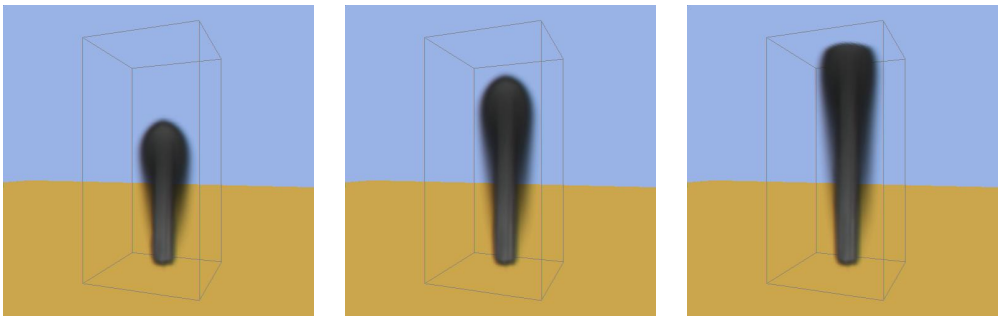
Test 3a: Heat Source

In this test we reuse the setup shown in figure 12.6. However, we disable the velocity source step, to disable the vent force, and enable the temperature source and thermal buoyancy steps. Hereby we insert a hot temperature source, that should cause the density to rise, in the same way as in test 2b.

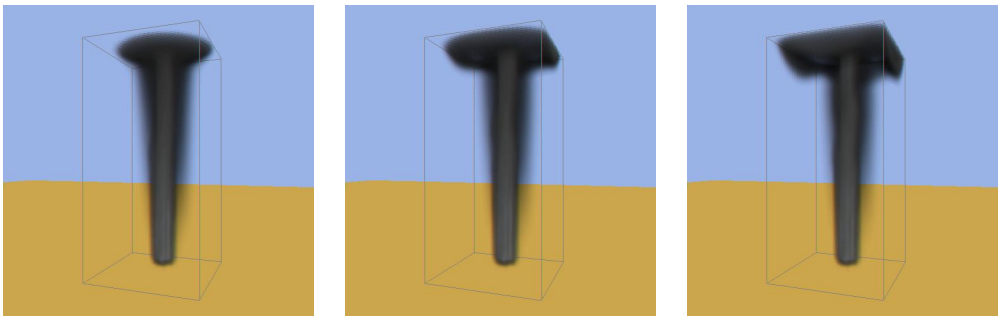
As shown in figure 12.10 on page 92, the temperature does infact cause the smoke to rise in a way similar to test 2b.



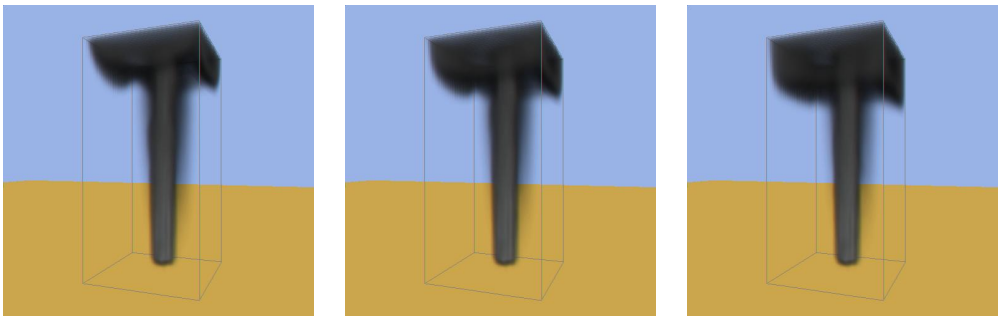
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.7: Images from test 2b.

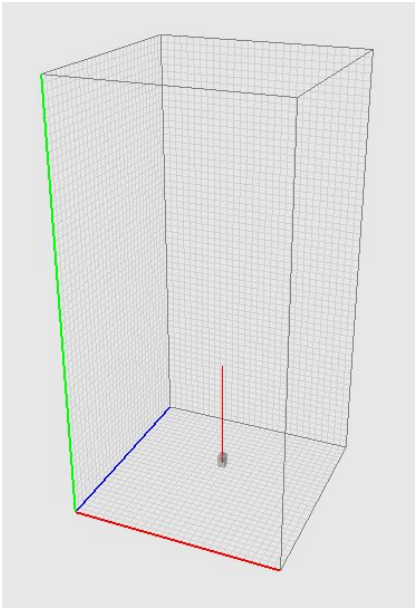


Figure 12.8: Setup for test 2c. This setup resembles the setup for test 2b in figure 12.6, only the density source is replaced with a smaller source. Density source and force is indicated by the gray boxes and the red line, respectively.

Test 3b: Cold Source

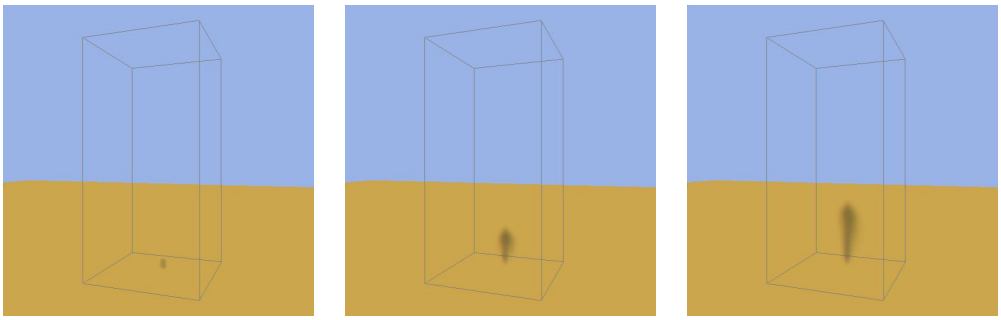
In this test, we turn test 3a upside down. Density and a cold source is inserted in the top of the simulation box, as shown in figure 12.11 on page 93. The enabled steps are: velocity advection, density source, density advection, temperature source, and thermal buoyancy. When running this test, the density should stream downwards, hit the ground and stream outwards and up the sides of the simulation box.

The result of this test is pictured in figure 12.12 on page 94. The animation resembles that of test 3a, only the smoke is falling, instead of rising, due to the lower temperature.

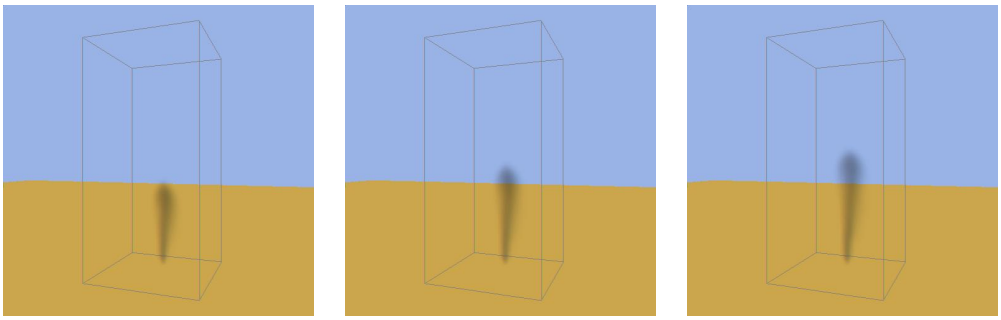
Test 3c: Hot Smoke

In this test we also enable advection of the temperature field. We use the setup shown in figure 12.13 on page 95, where a vent force, a density source, and a temperature source is inserted in the left side of the simulation box, thus, blowing hot smoke horizontally to the right. Since temperature and smoke are advected equally, the smoke should rise to rise due to thermal buoyancy, giving the impression of hot smoke.

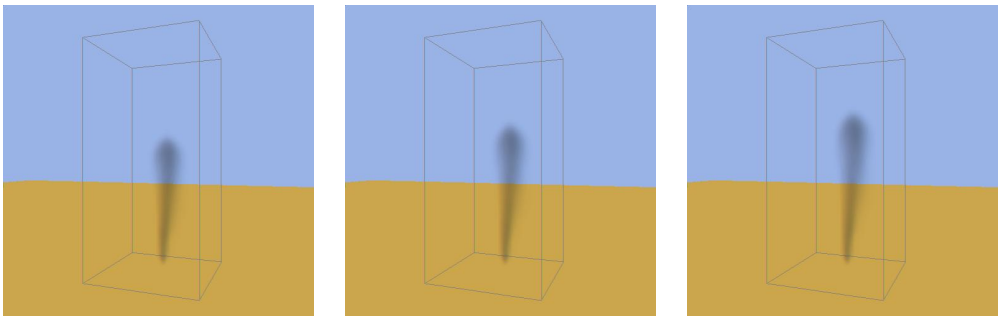
The running of this test is shown in figure 12.14 on page 96. We see that the smoke rises in a smoothly curved stream, giving the sense of the smoke having a temperature.



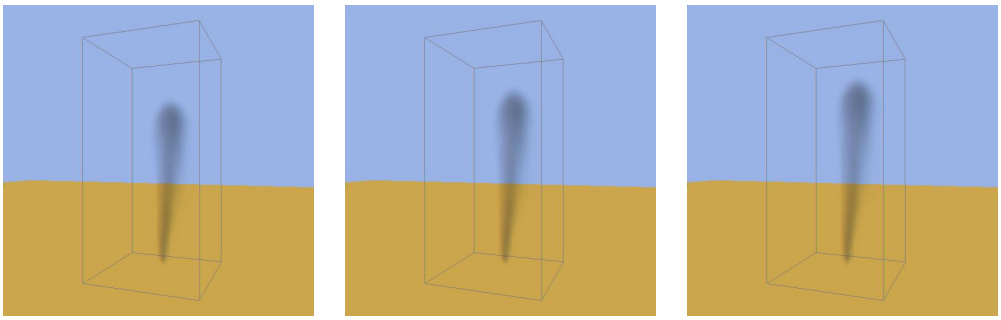
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

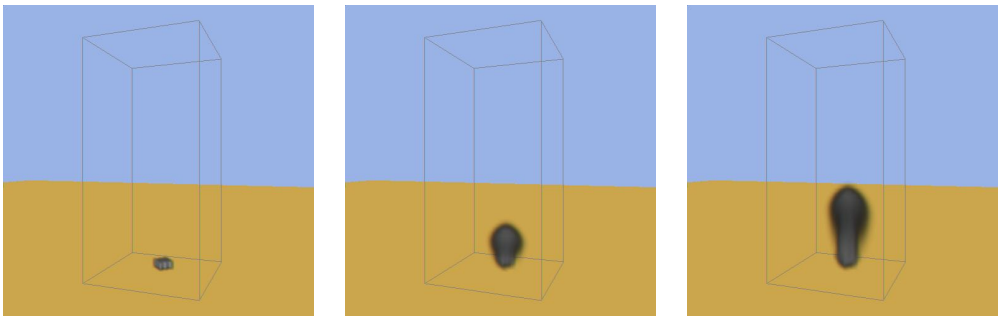


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

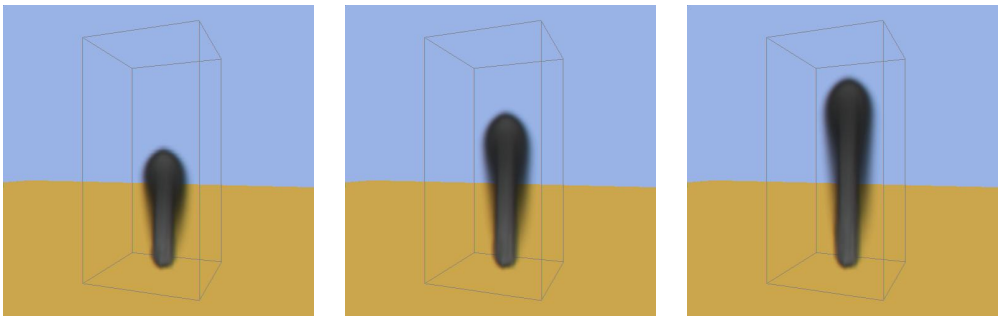


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

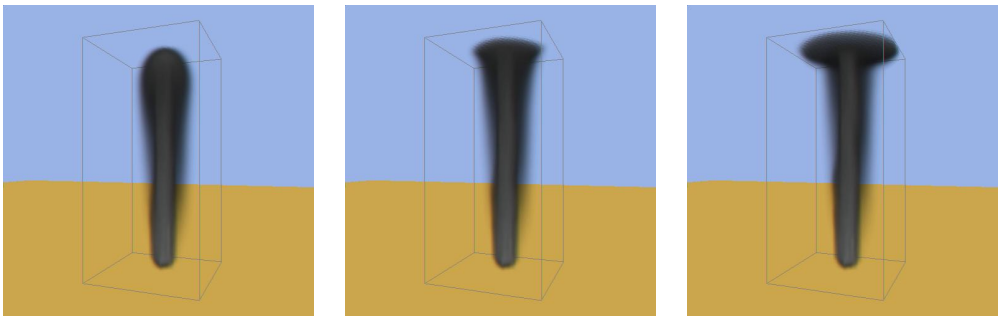
Figure 12.9: Images from test 2c.



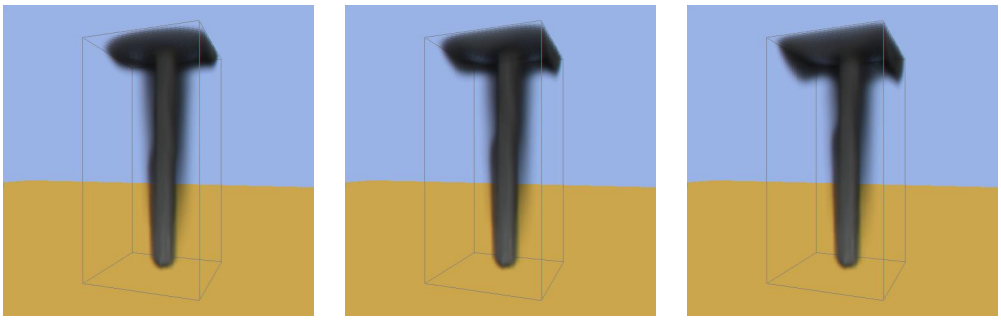
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.10: Images from test 3a.

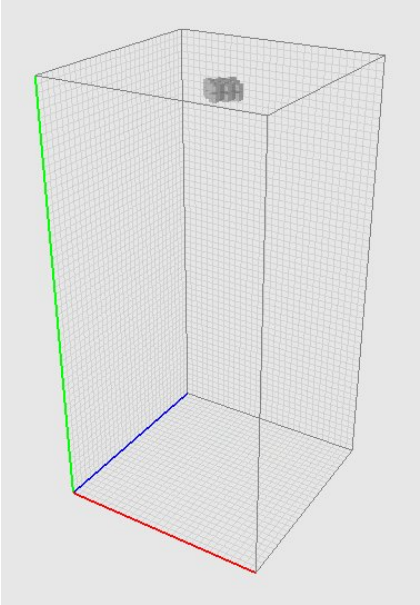


Figure 12.11: Setup for test 3b. Density and a cold source is inserted in the top of the simulation box.

12.2.4 Test 4: Vorticity Confinement

Since vorticity confinement is not based on a physical feature of fluid motion, all we can do to evaluate the correctness of the implementation, is to relate the effects to the supposed behavior.

Test 4a: Vorticity Confinement

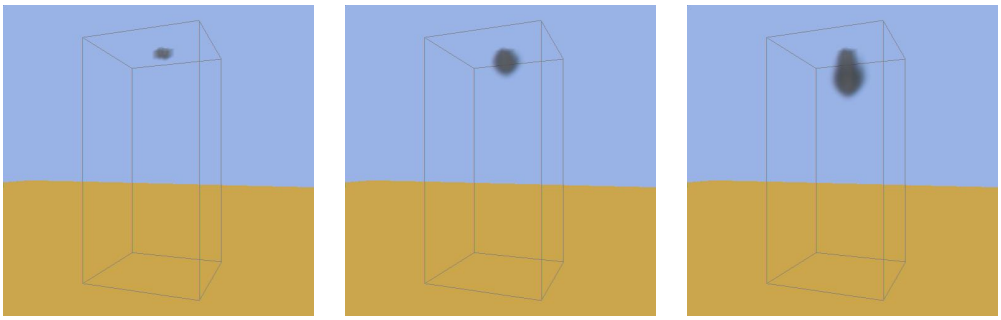
This test reuse the setup shown in figure 12.6 on page 88. We now enable the vorticity confinement step (so all steps are now enabled), which should cause the rising smoke to be more disturbed, and thus, more smoke-like than the smooth streams of previous tests.

As can be seen in figure 12.15 on page 97, swirls are added to the smoke stream, and the stream is widened on its way to the top of the box. We conclude, thus, that the vorticity confinement does add the effect we expected.

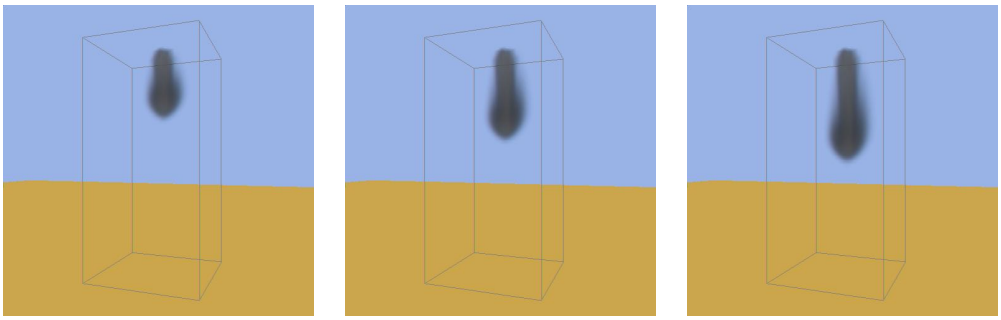
Test 4b: High Vorticity Confinement Control Parameter

In this test we use the same setup as test 4a (see figure 12.6 on page 88), but we increase the confinement control parameter, creating an even more disturbed smoke stream.

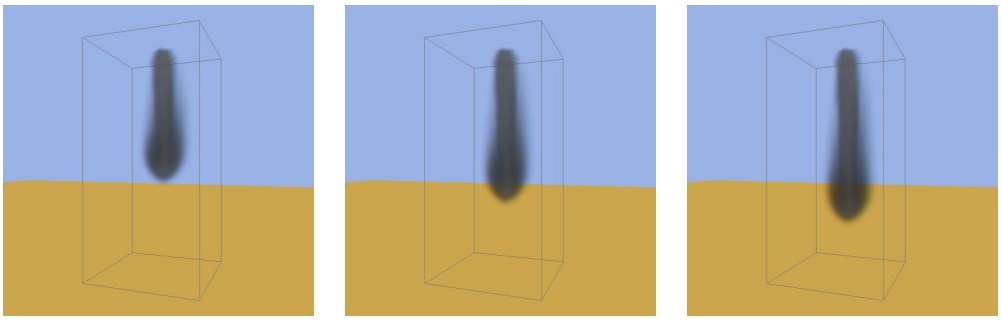
As expected, figure 12.16 on page 98 shows that the smoke does in fact become more chaotic and swirling.



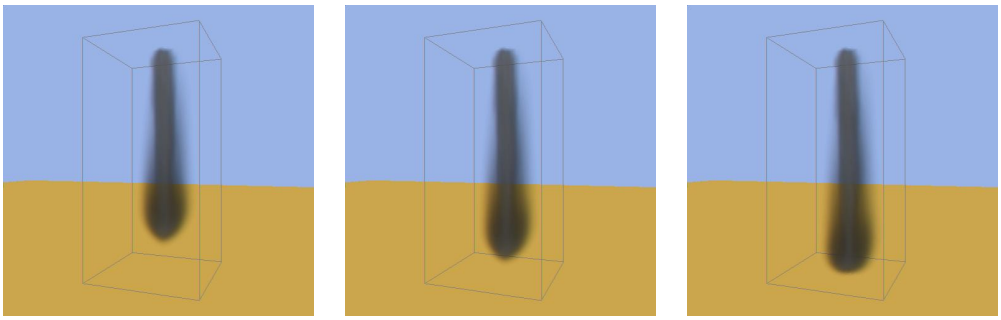
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.12: Images from test 3b.

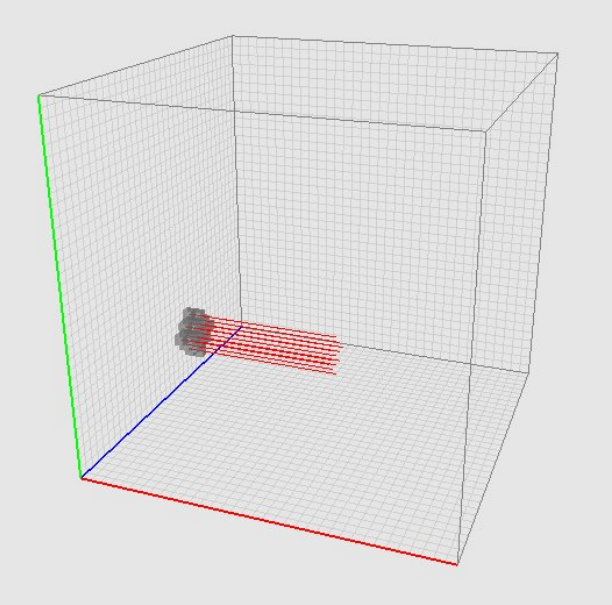


Figure 12.13: Setup for test 3c. Hot smoke is blown horizontally from left to right.

12.2.5 Test 5: Boundaries

Since the handling of internal boundaries in the solver is very simple, these tests should be seen as a clarification of its capabilities.

Test 5a: A Simple Object

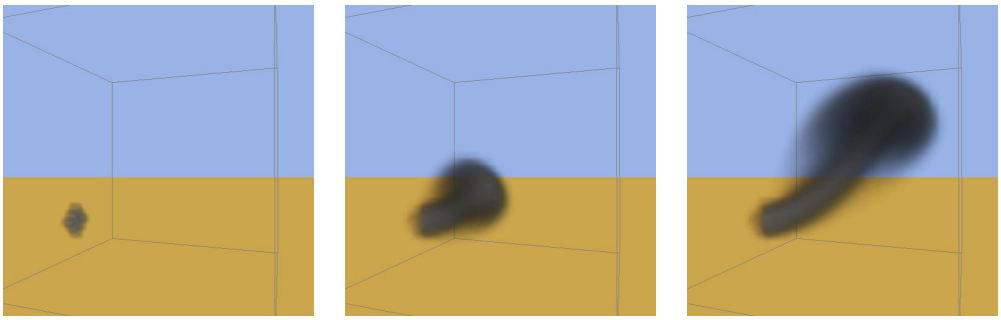
We first introduce a simple, solid object into the fluid. The setup is shown in figure 12.17 on page 99. At this point, we have tested all the solver steps, so we enable them all. The smoke should advect upwards until reaching the object. It should then *curl* around the object, and form a wider, less dense stream above the object.

From the results of the test, shown in figure 12.18 on page 100, we see that the simple method actually works very well with a simple object. The smoke rises to the object, it is forced around the object, and continues upwards with a curling behavior. When comparing with the results of test 4a, a slight volume-loss around the object is noted. However, when observed outside this context, the effects is better than we hoped for.

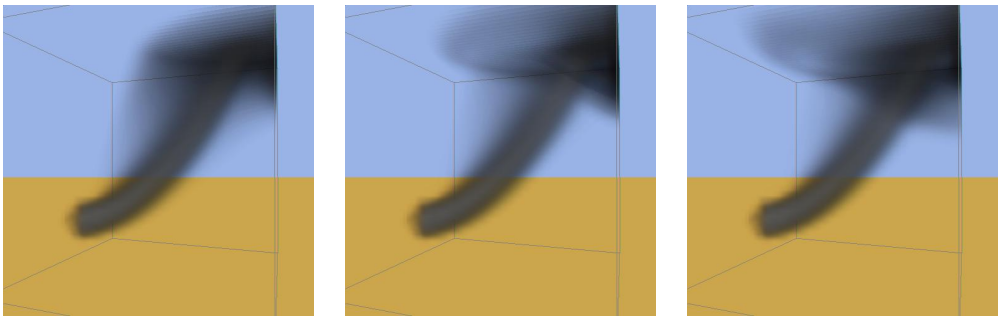
Test 5b: A Thin Object

As noted in section 9.2 the co-located representation traditionally has a problem with obstacles spanning only one cell. Thus, the simple handling of internal boundaries should be tested in this regard. For this, the scene shown in figure 12.19 on page 101 is used. Again, we enable all the solver steps. A result similar to test 5a is expected. Special attention should be made, that the smoke does not rise up through the plate.

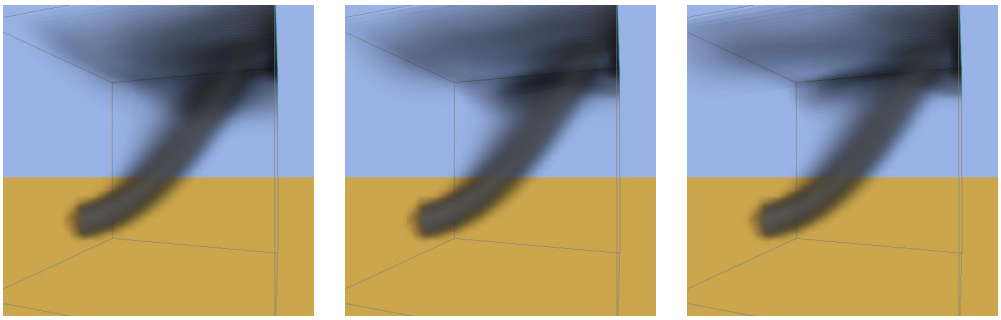
The result is show in figure 12.20 on page 102. We see that the volume-loss related



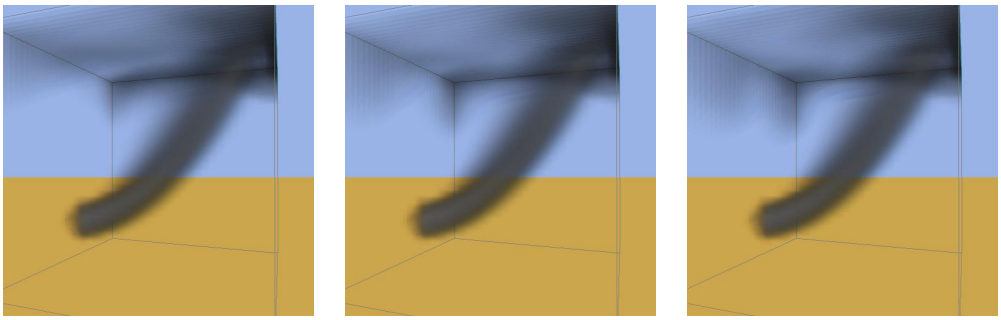
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

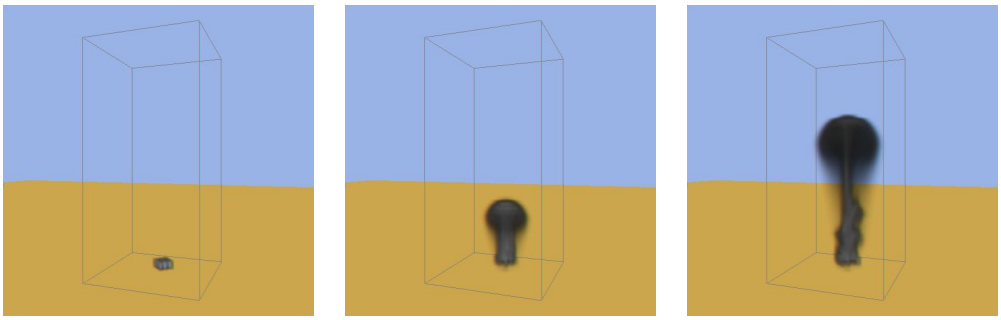


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

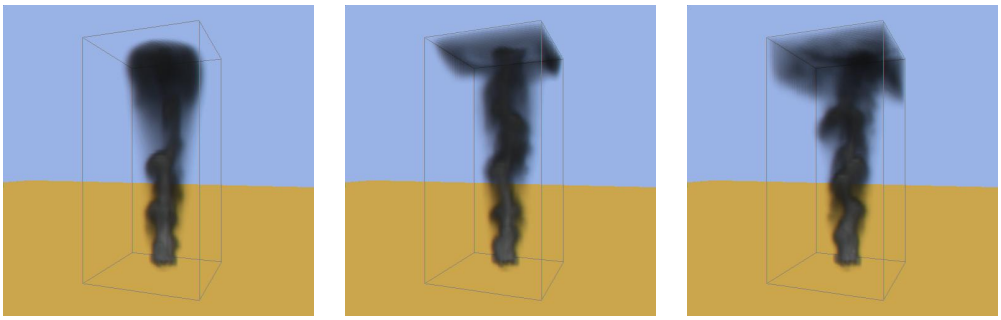


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

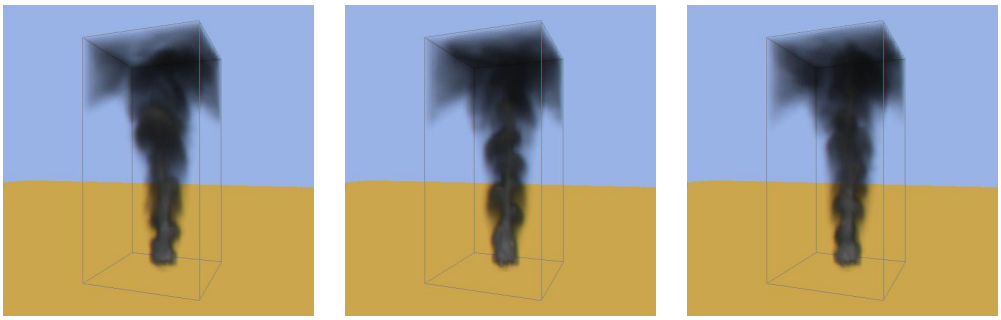
Figure 12.14: Images from test 3c.



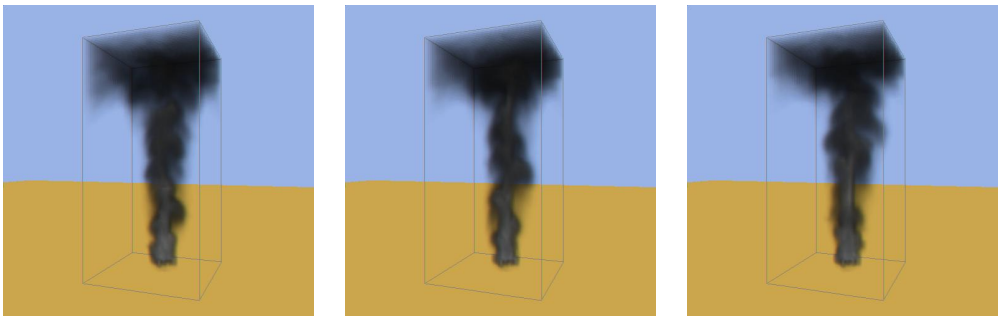
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

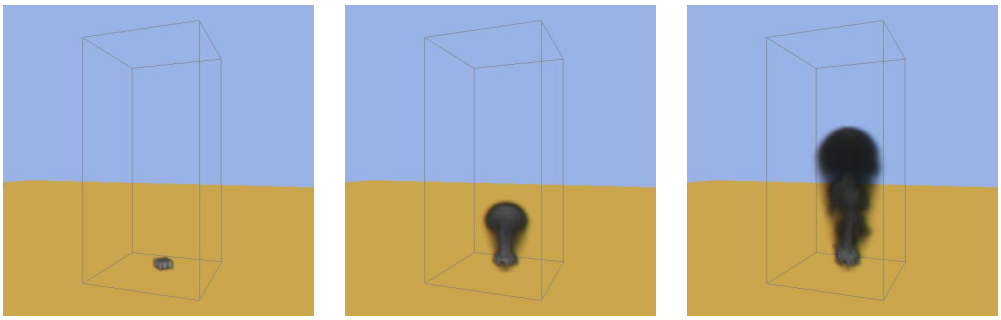


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

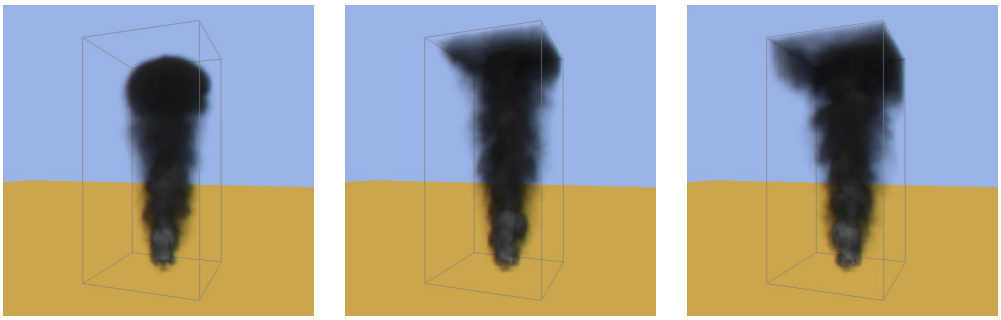


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

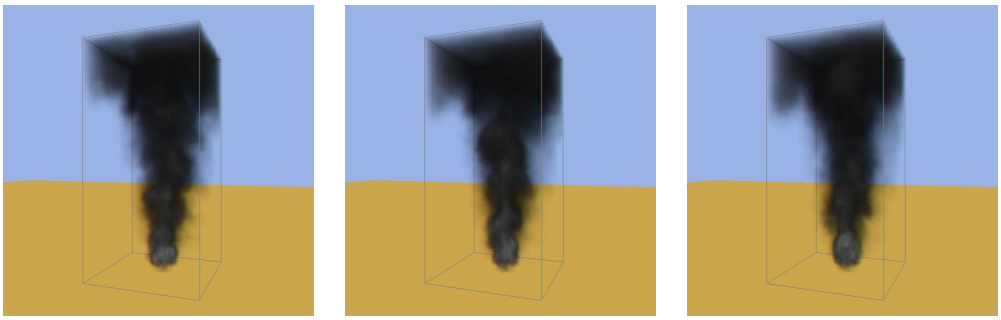
Figure 12.15: Images from test 4a.



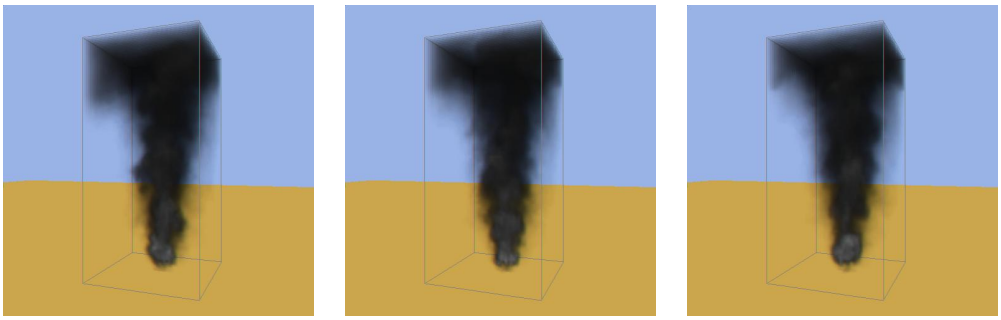
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.16: Images from test 4b.

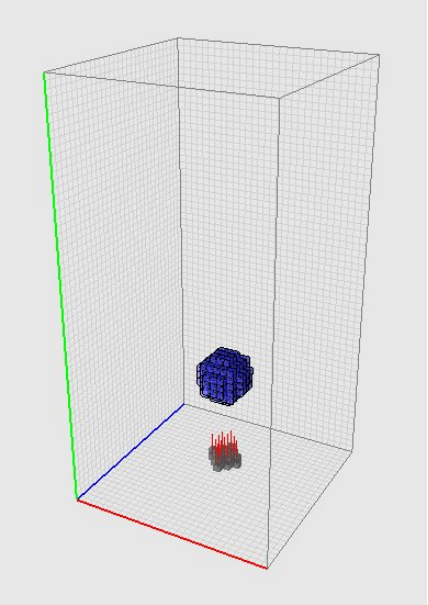


Figure 12.17: Setup for test 5a. A solid, voxelized, spherical object is inserted above a hot smoke source, similar to that used in earlier tests.

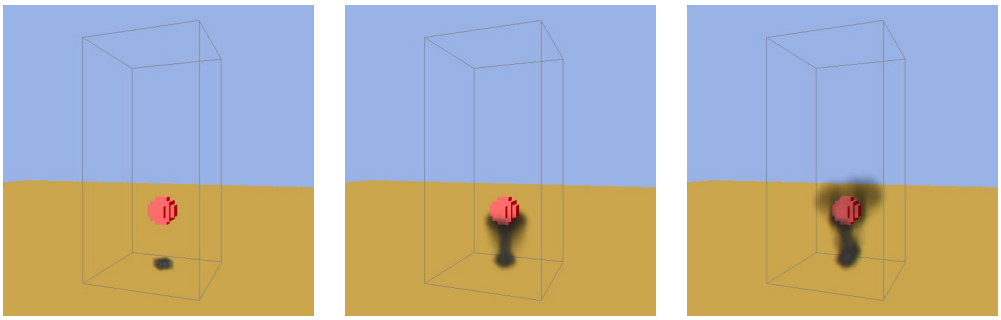
to internal boundaries is more expressed in this test; in fact it is so severe that it appears disturbing to the animation. The primary reason for this is the way the density field is handled in boundary cells: density values are, as velocity values, set explicitly to 0. Since some of the density is thus deleted, this is an obvious reason for the volume-loss. In [10], this is fixed by setting density values in the boundary cells adjacent to fluid cells to the average of the neihboring values. In this way density is actually represented inside the object, but the volume-loss *is* deminished.

At first sight, it seems that density does not move through the boundary. A closer look, however, reveals that small amounts of density do actually tunnel through the obstacles. If we observe a tile from the velocity field, as shown in figure 12.21 on page 103, the penetration is very clear. It comes as no mystery, though, because of the following two flaws in the handling of obstacles:

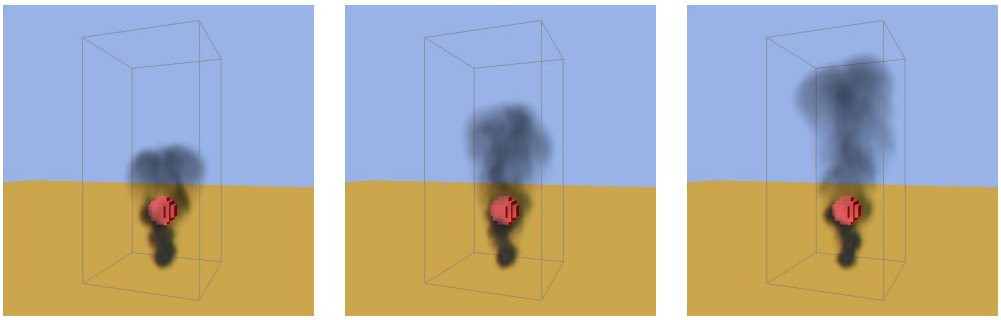
- Advection into or through the obstacles is not prevented in the advection step. Thus, large velocities or large time steps will allow advection across an obstacle cell.

- When iterating the pressure field, boundary conditions are only set for the volume boundaries – not the obstacles. Thus, the pressure is not increased to keep the fluid from entering the obstacle cells.

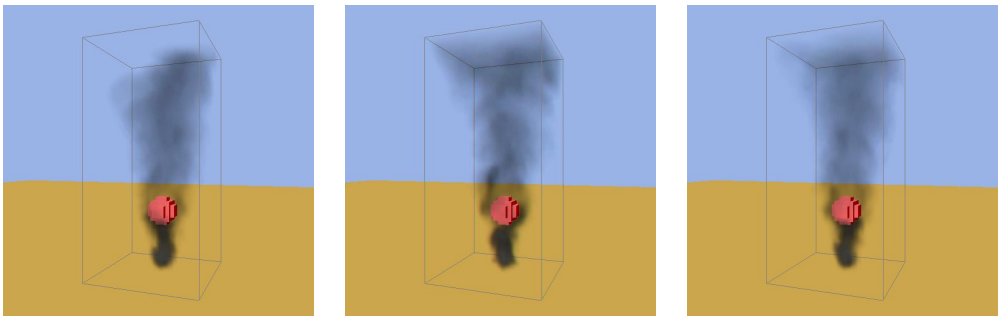
As seen in test 5a, the effect of these flaws can be deminished by making the objects span over more cells in each direction.



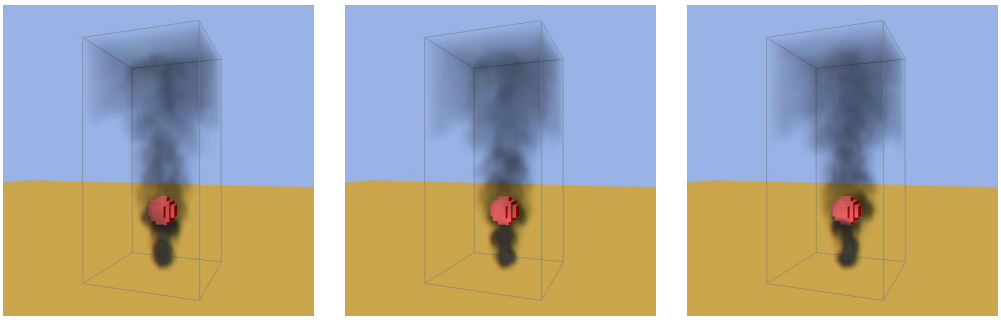
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.18: Images from test 5a.

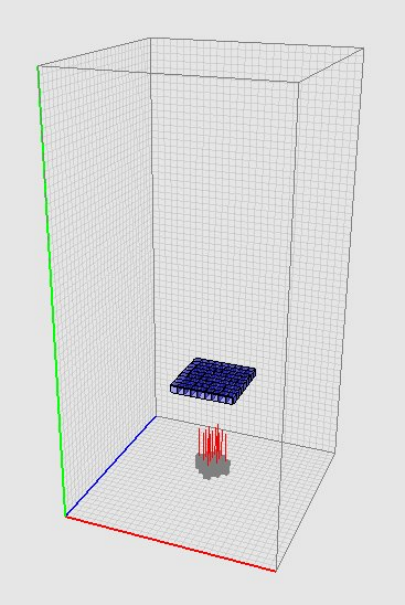


Figure 12.19: Setup for test 5b. A one cell high plate is placed above a hot smoke source, similar to the one used in earlier tests.

12.2.6 Test 6: Texture Precision

In these test we will run the same simulation with different texture map precisions, to get a feeling of how much influence the precision has on the precision and frame rates of the simulations. For all tests we will use the setup from figure 12.6 on page 88, and all simulation steps will be enabled.

Test 6a: Low Precision

This test uses the lowest precision available in OpenGL texture maps. All fields are defined using the OpenGL internal format `GL_RGBA8`, which causes texture maps to use 8-bit fixed-point precision. For interpolation we use the standard OpenGL filtering, `GL_LINEAR`. It is uncertain if this precision is enough for simulating fluids.

The result of this test is shown in figure 12.22 on page 104. The smoke is moving in a totally wrong direction, so something is obviously wrong. On the other hand, the smoke stream, although crooked, still shows some of the behavior seen in previous test, so we cannot simply write everything off as just imprecision.

An immidiate guess would be that the scaling of the 8-bit values, as mentioned in section 11.2.3, is not symetrical around zero. This would cause a round-off error in every frame, thus, quickly accumulating to divergent behavior. The smoke is diverging in both the positive x and z directions, and presumably in positive y direction as well, which supports this idea. Unfortunately, we have not had the time to experiment with this, so the hypothesis will be left without proof.

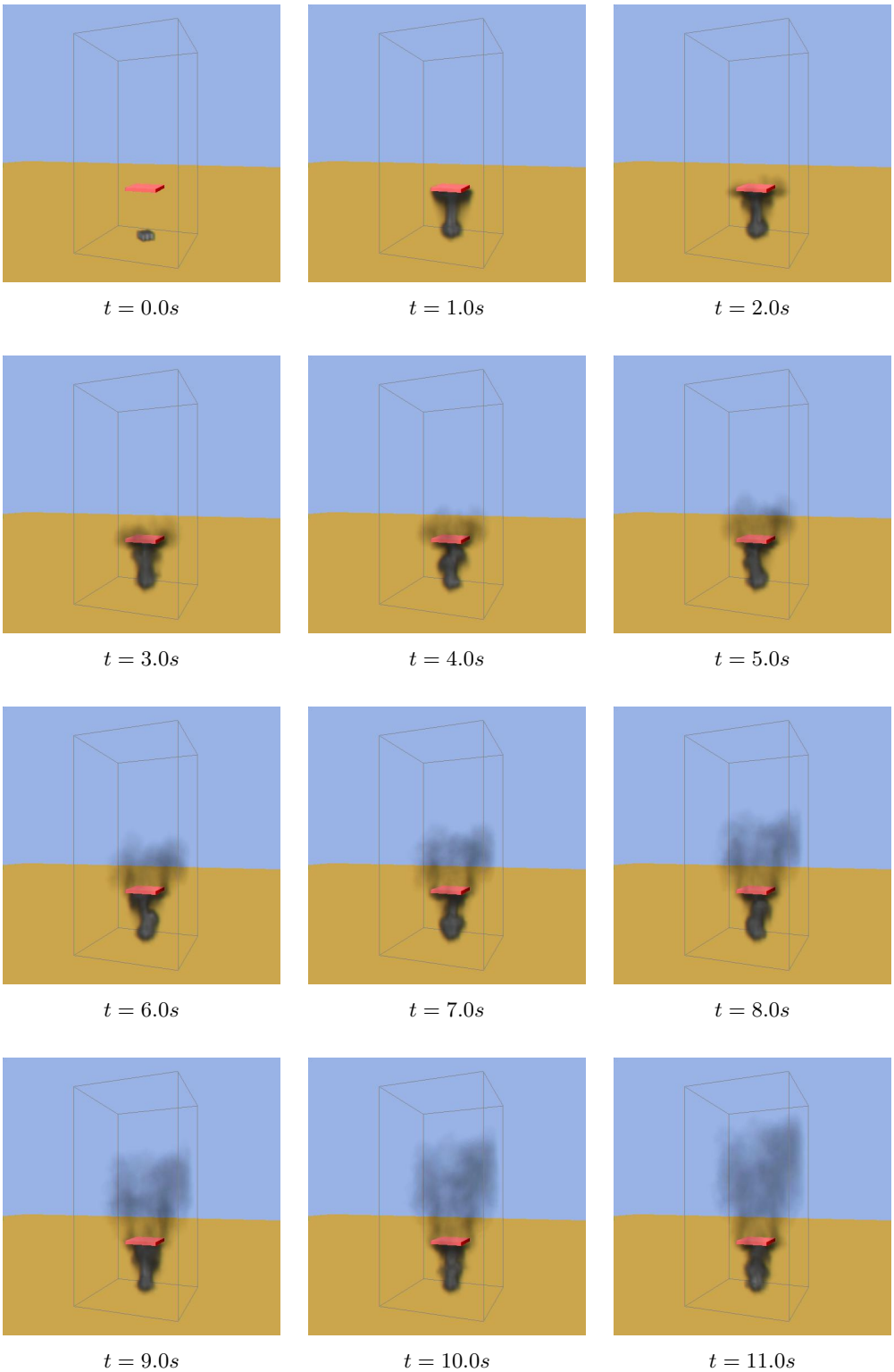


Figure 12.20: Images from test 5b.

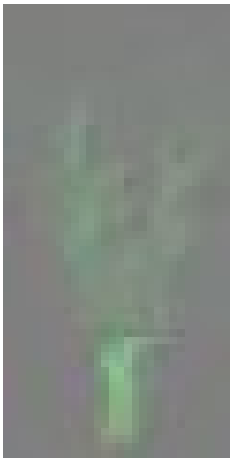


Figure 12.21: A tile from the velocity field from test 5b. The green component indicate velocity in the y direction (vertical). The velocity is clearly advected through the obstacle.

Table 12.1 shows, as expected, that the frame rate is a little higher in this test than the rest of the test, although the gain is not very impressive, considering the visual output.

Precision	Running Time	Number of Frames	Avg. Frame Rate
8-bit (test 6a)	23.09 seconds	401	17.36 fps
16-bit (test 6b)	24.89 seconds	401	16.11 fps
16-bit (test 6c)	25.62 seconds	401	15.65 fps
16-bit (test 6d)	25.45 seconds	401	15.76 fps
32-bit (test 6e)	33.38 seconds	401	12.01 fps

Table 12.1: Time meassurements of the tests 6a-f.

Test 6b: Medium Precision 1

In this test we increase texture map precision to 16-bit fixed-point, by using the internal format `GL_RGBA16`. Interpolation is still done using `GL_LINEAR`.

The images in figure 12.23 on page 105 show the result of this test. We see that the divergent behavior of the previous test is removed, and nicely rising smoke is presented. As expected, table 12.1 shows that the frame rate is a little lower than in the previous test.

Test 6c: Medium Precision 2

In this test we keep the precision as in test 6b, but we change the interpolation method, to the our own implementation. Frame rates are expected to be lower in this test, than in test 6b, because of the interpolation method.

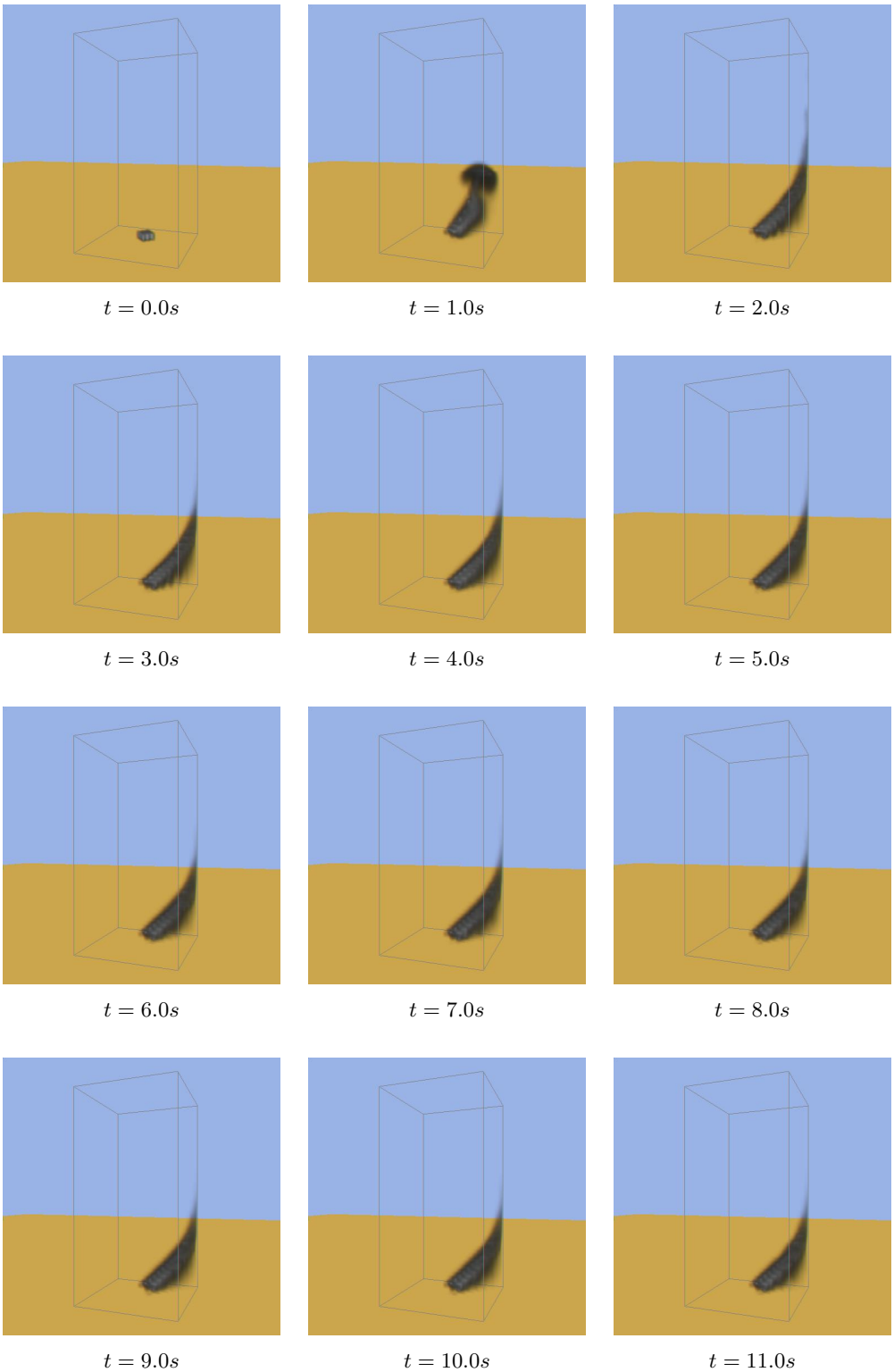
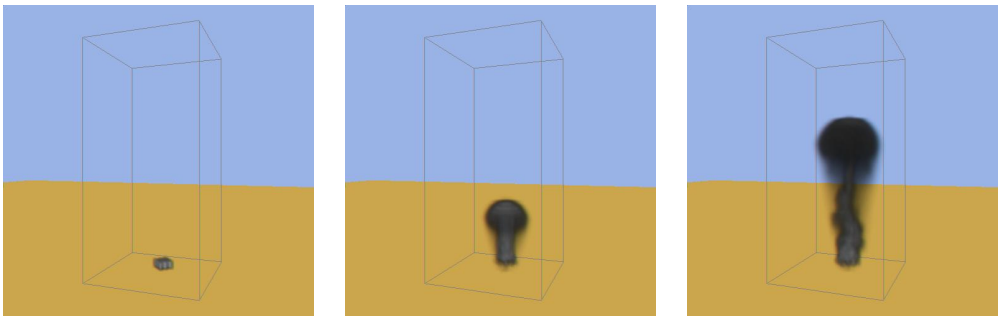
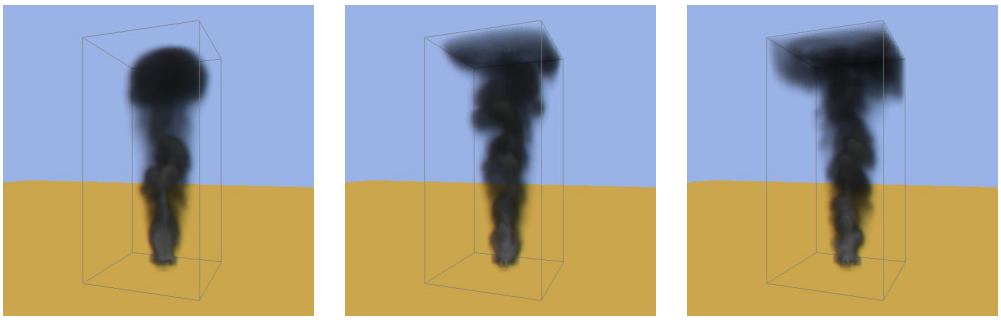


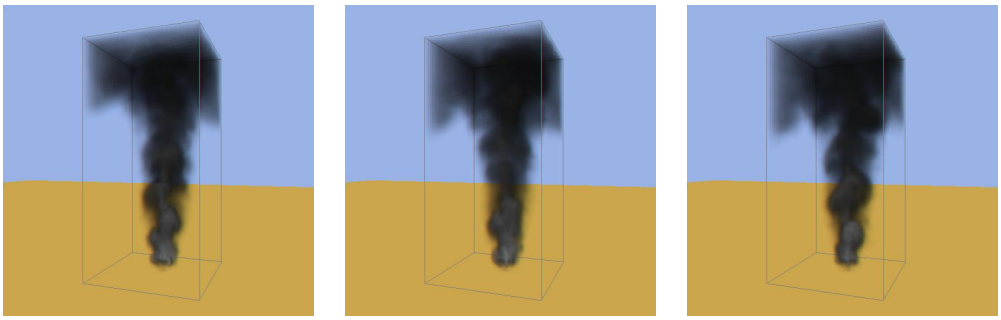
Figure 12.22: Images from test 6a.



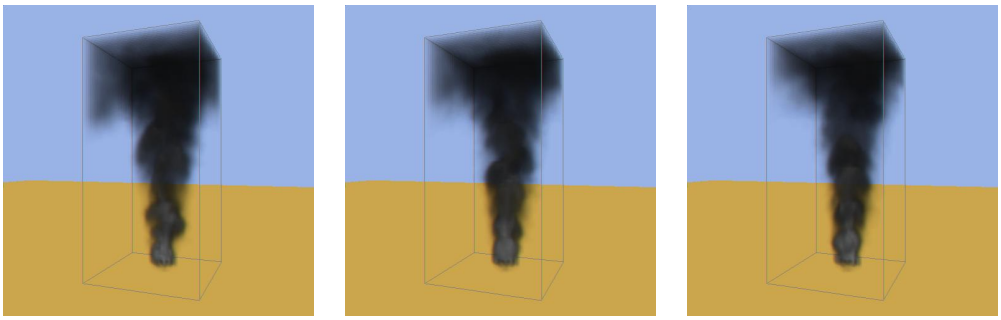
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.23: Images from test 6b.

From figure 12.24 on page 107 we see that using the trilinear interpolation implemented in the advection fragment shader produce almost the exact same result as in test 6b. Table 12.1 confirm the presumpsion about the frame rate.

Test 6d: Medium Precision 3

In this test we change the texture map precision to `GL_RGBA_FLOAT16_ATI`, which is a floating point texture format, used on ATI graphics adapters. When switching to this format, we can no longer use the `GL_LINEAR` interpolation method, thus we use our own implementation.

Since `GL_LINEAR` interpolation does not work on the density field either, the density in this test will seem *pixelized*. The frame is expected to drop even more, because of the higher precision.

The fallout of this test, shown in figure 12.25 on page 108, is a bit peculiar, and not what was expected. The smoke stream is crooked, as it was in test 6a, but not to the same extend, and this time in negative directions of x, y, and z. What causes this is a bit of a puzzle, but again we point at the up and down scaling of values, when reading and writing texture values.

Test 6e: High Precision

In this test we use the highest precision available on current GPUs, 32-bit floating point. Again, we are forced to use our own implemented interpolation. From this test we expect the best simulation, but also the lowest frame rates.

In figure 12.26 on page 109 we see that the increased precision does help on the problem experience in test 6d. We do not, however, get any noticable difference in the images. Since table 12.1 show that this configuration has the lowest frame rate of all the precision tests, the increased precision does not seem very attractive.

12.2.7 Test 7: Colored Density

In this test we demonstrate the capabilities, which come freely when representing the density field with a four-component texture map.

Test 7a: Transparent Smoke

We use the setup of figure 12.6 on page 88, but replace the density source with a more transparent one. In previous tests the alpha component were kept at $A = 1.0$. In this test we lower this to $A = 0.3$ and expect a more transparent smoke. The color of the smoke is set to black, in order to make the transparency more appearant.

The result is shown in figure 12.27 on page 110. Images from test 7a. Even though the color of the smoke has been set to black, we see that the smoke is very bright. As supposed, the smoke is very transparent; the bounding box of the simulation volume, as well as the brown/blue border in the background, can easily be seen through the smoke.

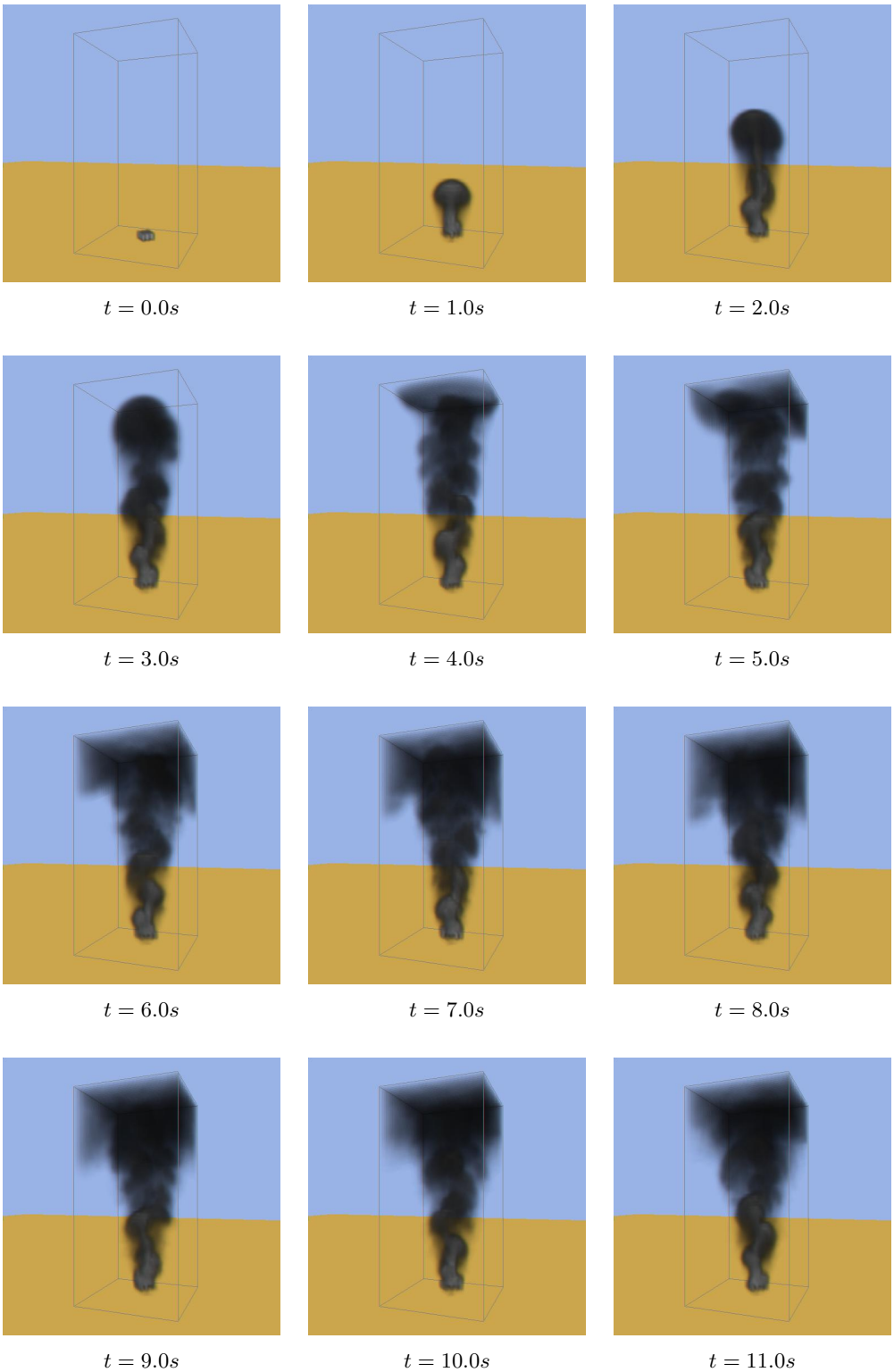
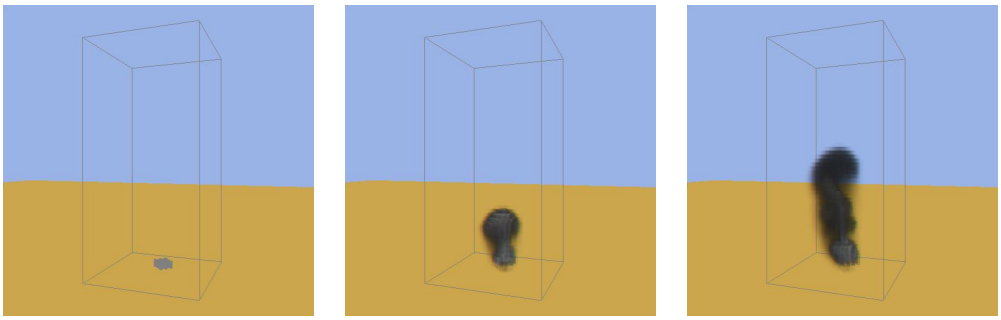
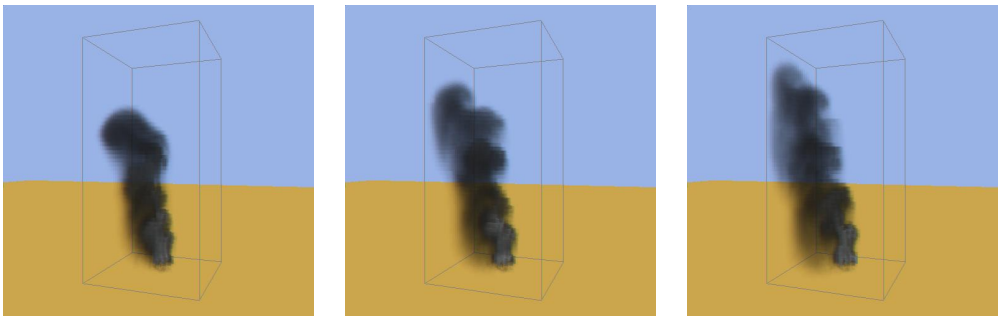


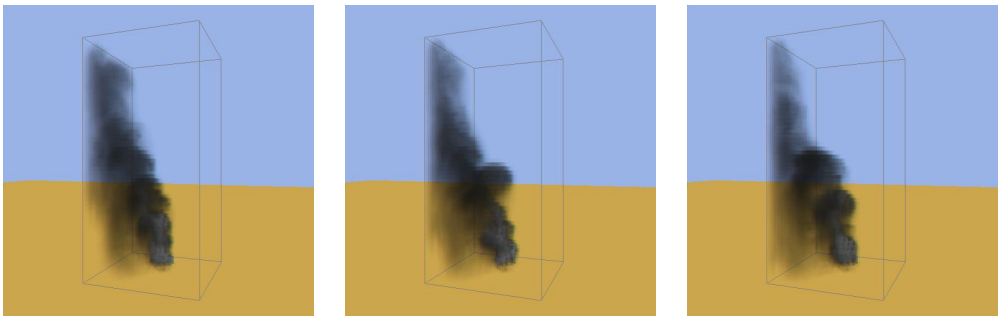
Figure 12.24: Images from test 6c.



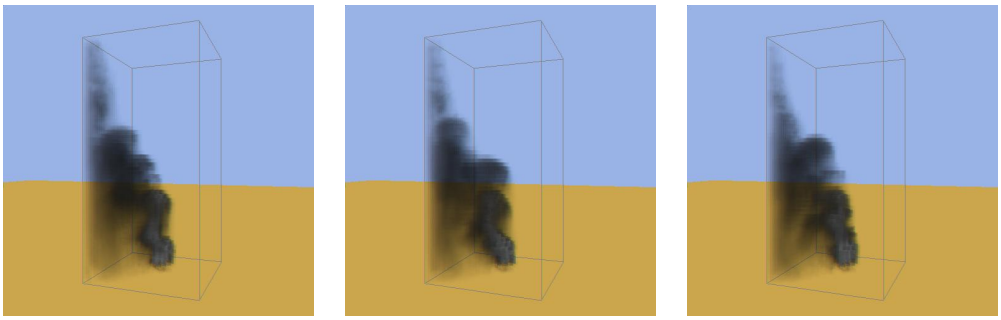
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.25: Images from test 6d.

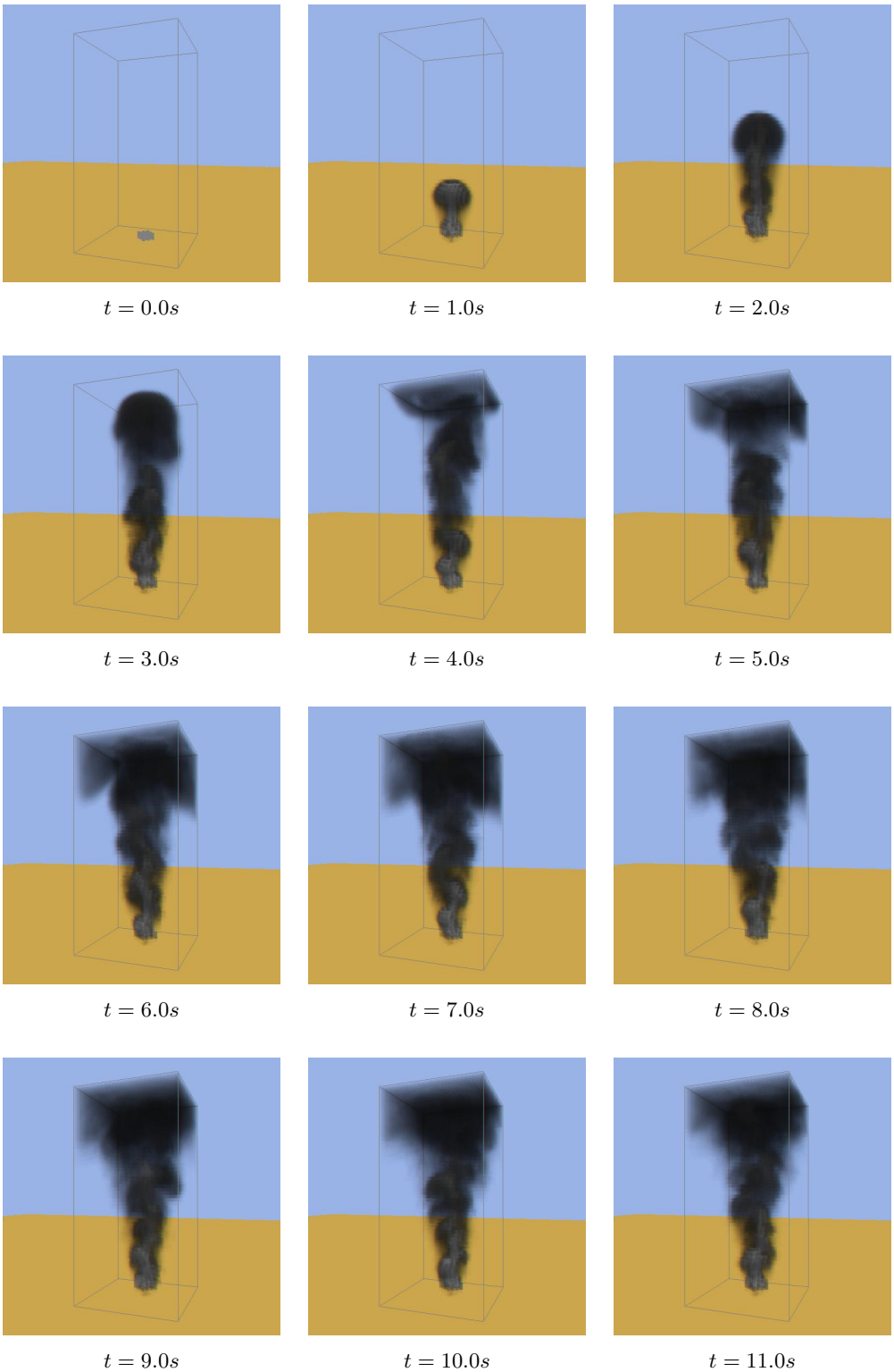
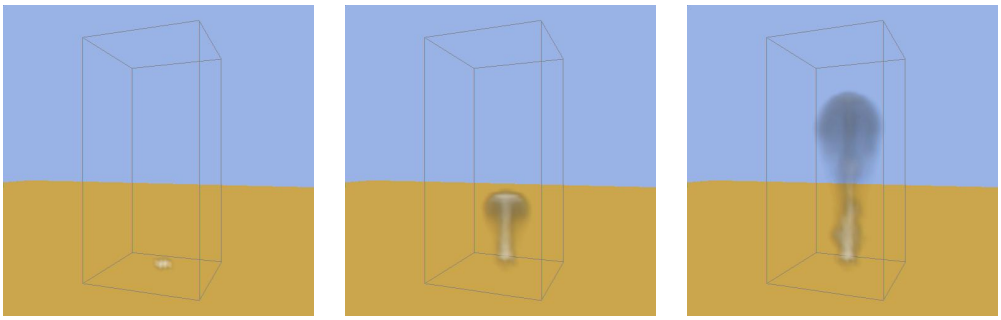
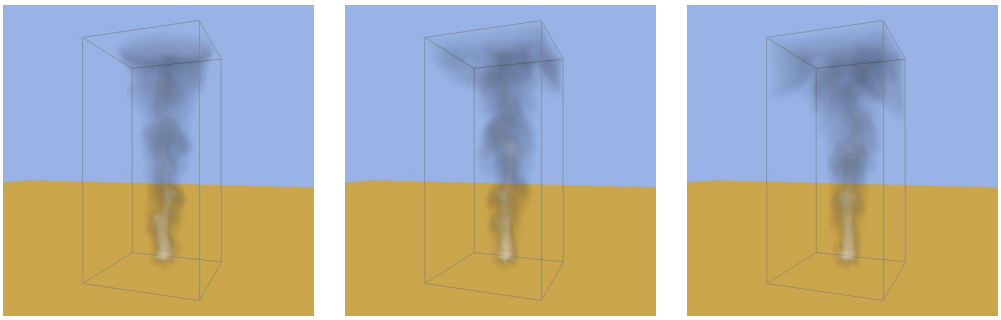


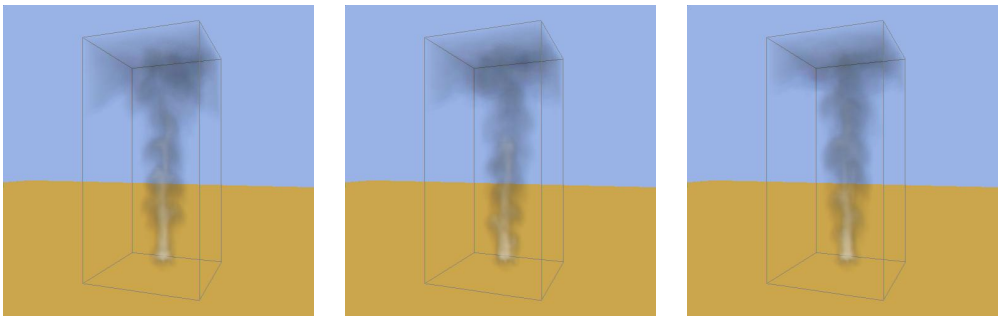
Figure 12.26: Images from test 6e.



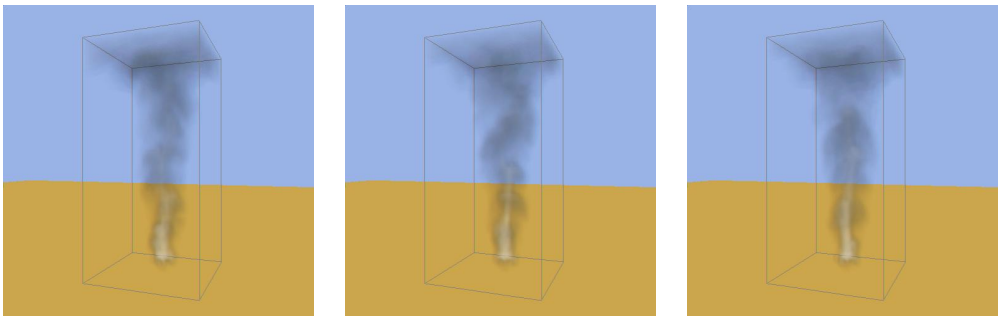
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.27: Images from test 7a.

Test 7b: Colored Smoke

Again we use the setup of figure 12.6 on page 88, but this time we replace the density with a source of colored density. The color of the smoke is set to $(R, G, B) = (0.0, 0.9, 0.65)$ (poisonous green). The alpha value is set to $A = 0.6$ to give a little transparency.

When representing smoke with a density field, we have to define a value for the color of "no smoke" – the base color. In previous tests we animated black and gray smoke, thus we set the smoke base-color to black and fully transparent ($(R, G, B, A) = (0, 0, 0, 0)$). In this test we define the smoke base-color to the same as the smoke color, except the base-color is made fully transparent ($A = 0$). Images from this test is shown in figure 12.28 on page 112.

Test 7c: Changing Colors

In this test we demonstrate the effect of changing the base-color of the smoke. The base-color is set to $(R, G, B, A) = (0.75, 0.15, 0.75, 0)$ (bright purple), which should cause the color of the smoke to change from poisonous green at the smoke source to bright purple when the smoke blends with the surrounding air. This effect is demonstrated in figure 12.29 on page 113.

Test 7d: Multi-Colored Smoke

The solver can control different colors of smoke sources per cell. This test demonstrates the use of two differently colored smoke sources and the mixing of colors when the the two smoke streams collide. The setup used is shown in figure 12.30 on page 114. To keep both smoke streams from blending into black, the smoke base-color is set to $(R, G, B, A) = (0.75, 0.15, 0.75, 0)$.

Images from this animation is shown in figure 12.31 on page 115. When the two smoke streams mix, the smoke is blended to the mixing color. It is noted that the border between both smoke streams and the surrounding air is the same color due to the common base-color.

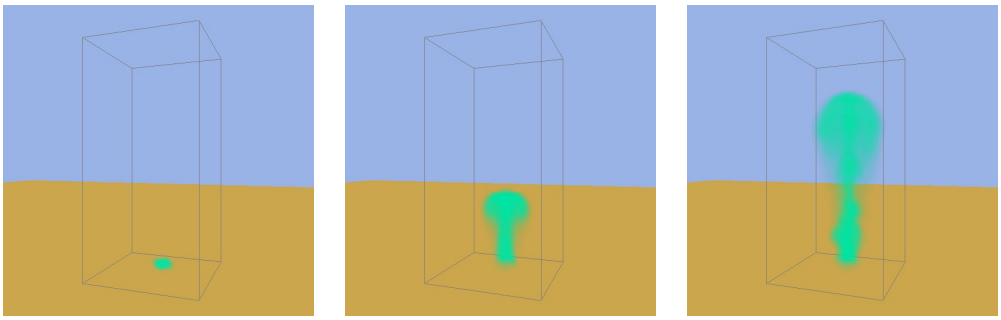
12.2.8 Test 8: Time and Stability

In all previous tests the time step has been fixed to $dt = 0.05$, corresponding to 20 frames per second. In this section experiments are made with different timesteps, to explore the stability of the solver regarding different time steps. For all tests the setup shown in figure 12.6 on page 88 is used.

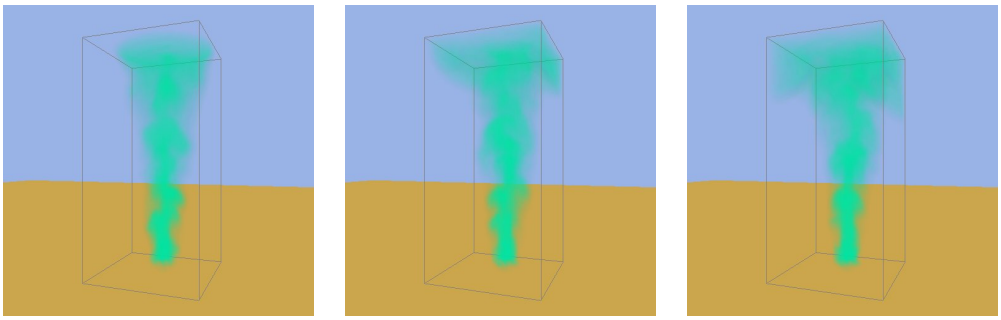
Test 8a: Small Time-Step

First the time step is set to $dt = 0.01$ corresponding to 100 frames per second to see if this actually produces smoother and more precise animations.

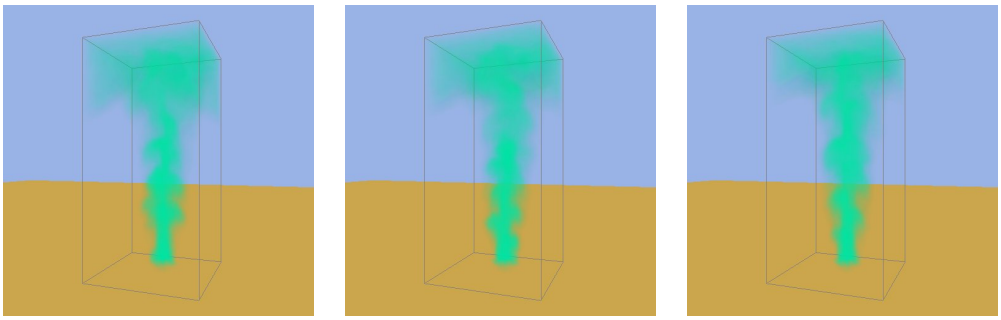
As can be seen in figure 12.32 on page 116, this test runs differently than the other tests using the same setup. The first thing to notice is that this simulation does not run in real-time, since the time step is a lot lower than the obtainable per-frame computation time. However, the simulation does not seem much more precise – only different. Despite



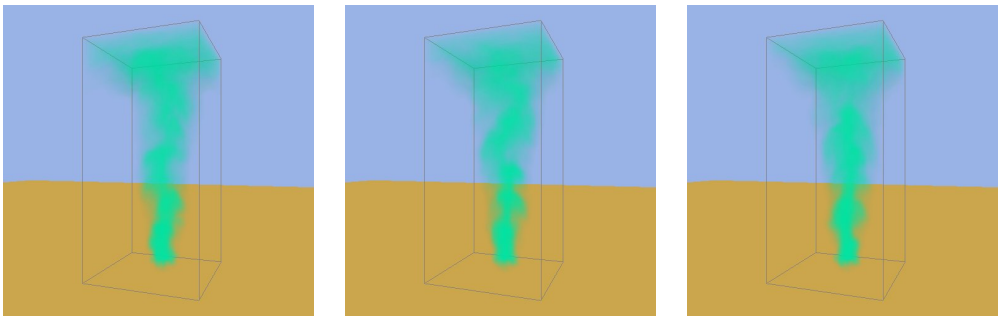
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

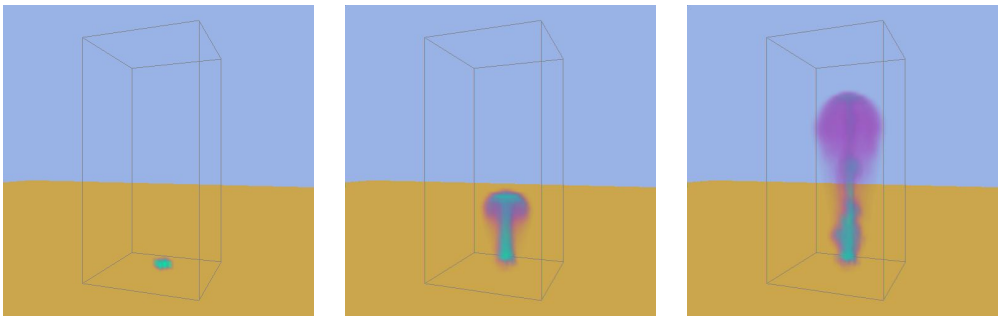


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

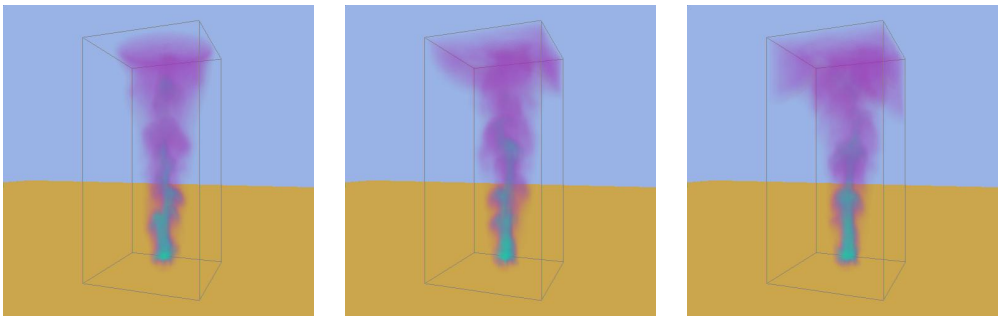


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

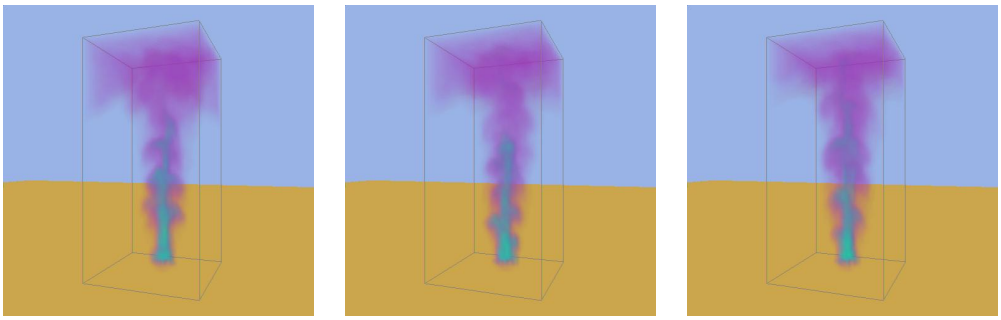
Figure 12.28: Images from test 7b.



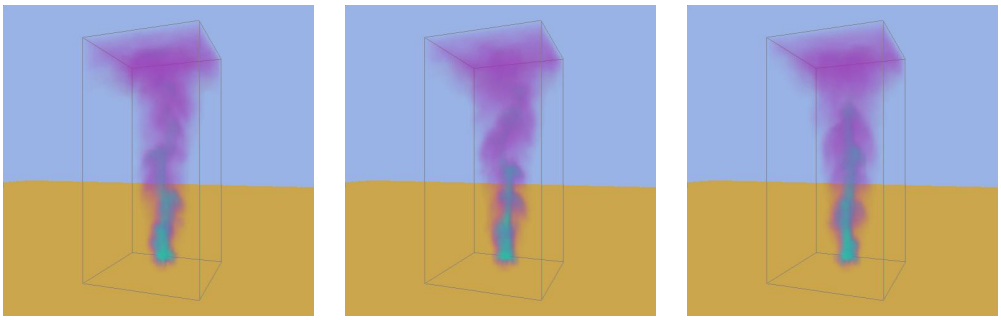
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.29: Images from test 7c.

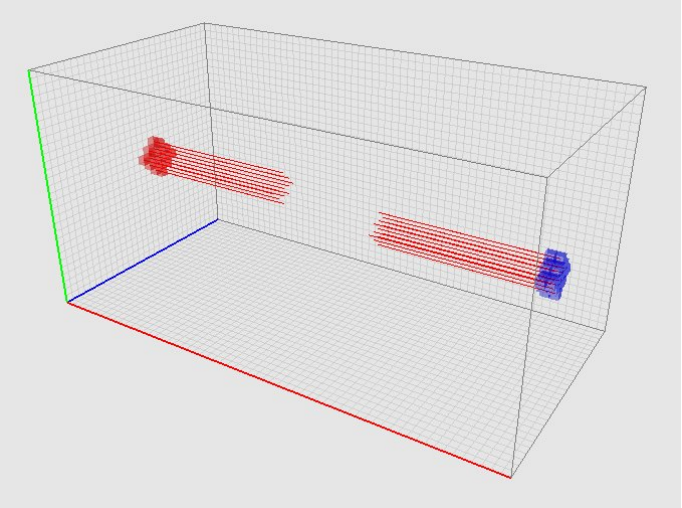


Figure 12.30: Setup for test 7e. Two smoke sources with different colors are set up in the opposite ends of the simulation box.

the small time step, the volume-loss experienced in earlier test is still apparent in the top of the simulation box. It may even seem that the small time step restrains the smoke from evolving.

Test 8b: Big Time-Step

The time step is now set to $dt = 0.1$ to see if the simulation can even be performed with this time step.

We see, in figure 12.33 on page 117, that the simulation does run nearly in the same way as in earlier tests, except for the chunky animation and some jittering in the density field near the smoke source. The volume-loss is a little more expressed, but overall the animations is better than we dared to hope.

Test 8c: Very Big Time-Step

In this test, we demonstrate that even though the semi-Lagrangian advection scheme is stable for large time steps, the simulation can *blow up* if too large time steps are taken. The time step is set to $dt = 0.3333333$, corresponding to 3 fps, and the result is shown in figure 12.34 on page 118.

12.2.9 Test 9: Time and Grid Sizes

The obtainable frame rate is very dependent of the size of the grid – a larger grid yields a lower frame rate. These tests are meant to clarify the relation between the size of a grid and the frame rate of the simulation with that grid. The previous tests using a grid resolution of $30 \times 60 \times 30$ is used as reference.

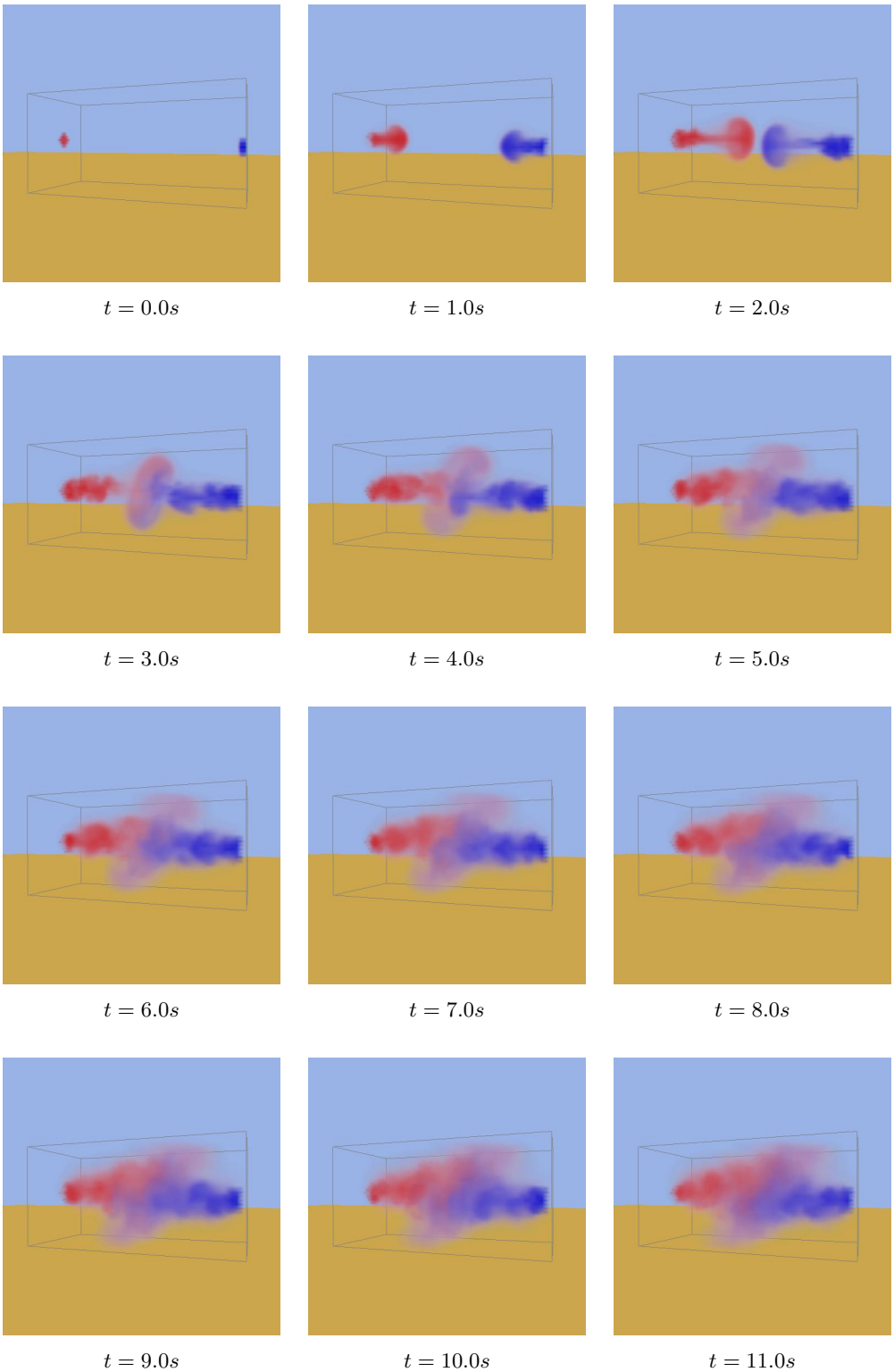
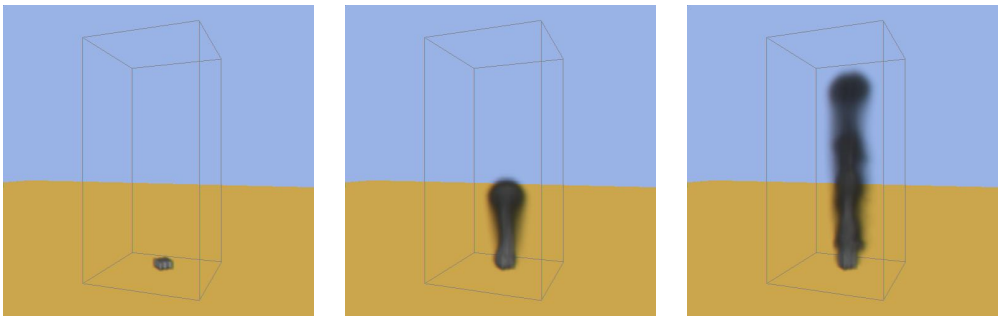
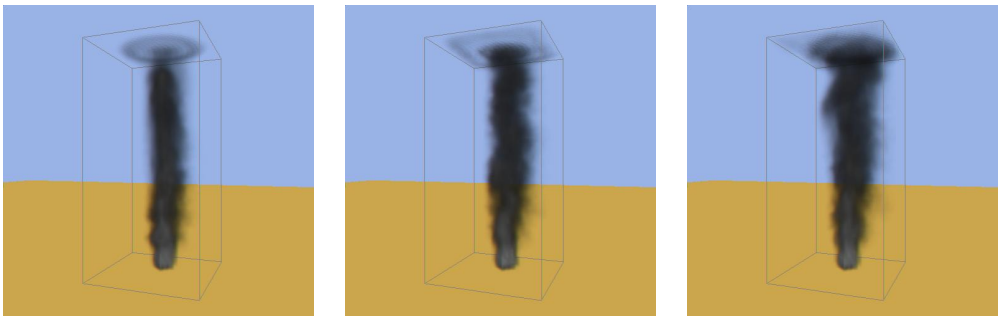


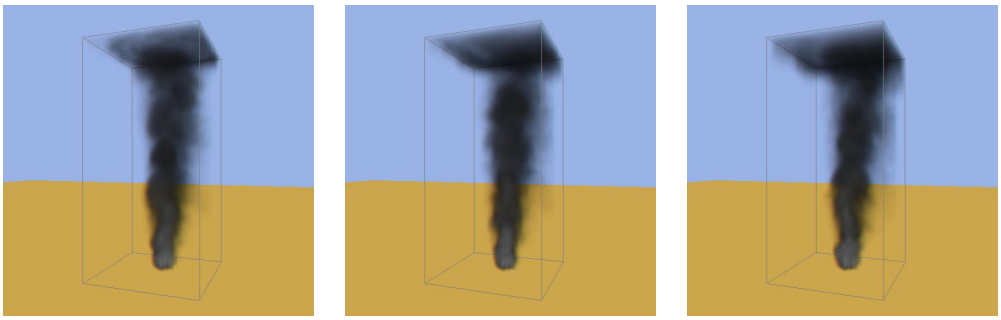
Figure 12.31: Images from test 7d.



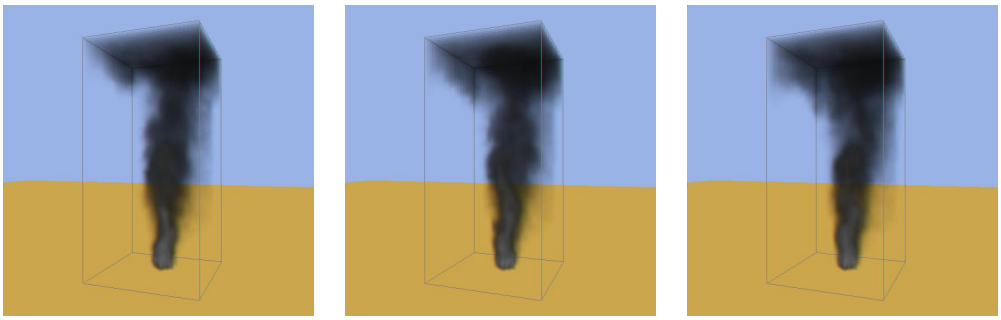
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

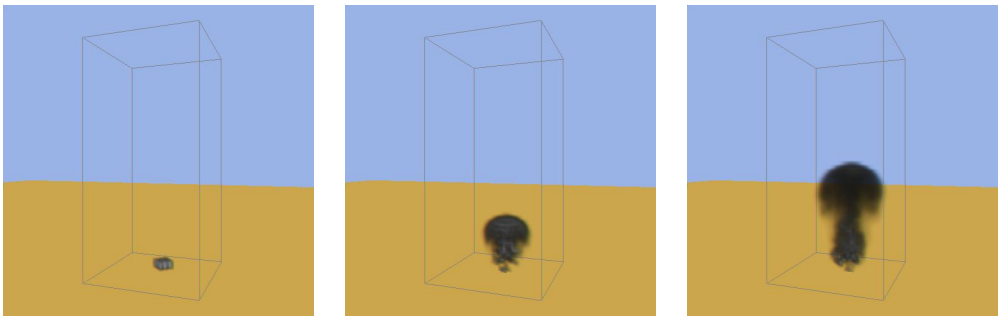


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

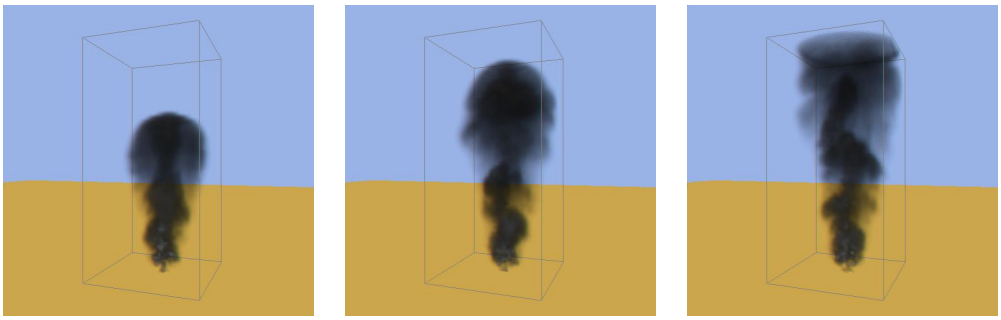


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

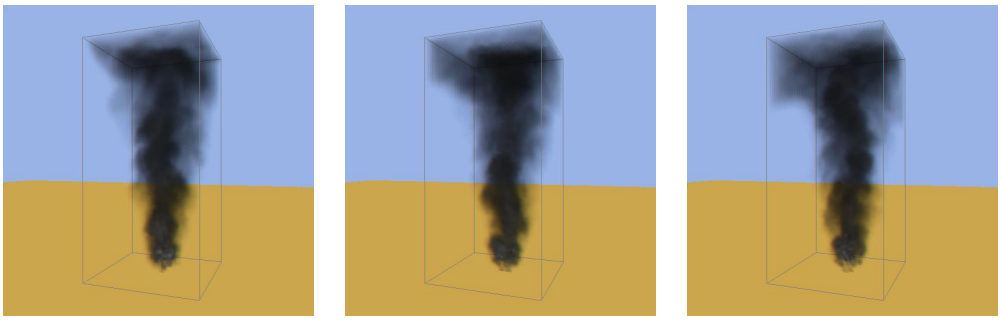
Figure 12.32: Images from test 8a.



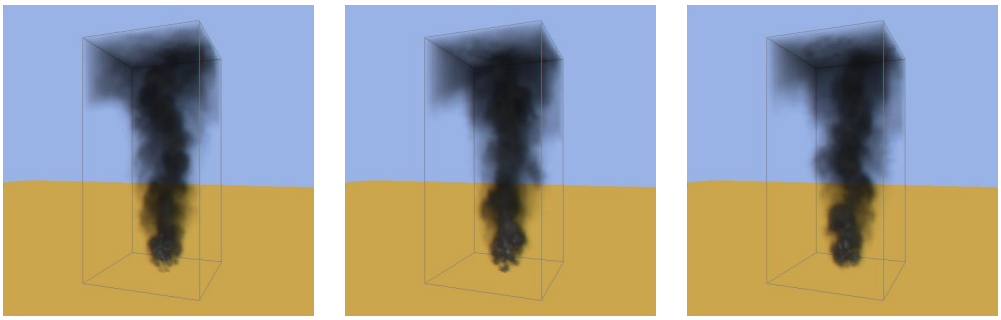
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

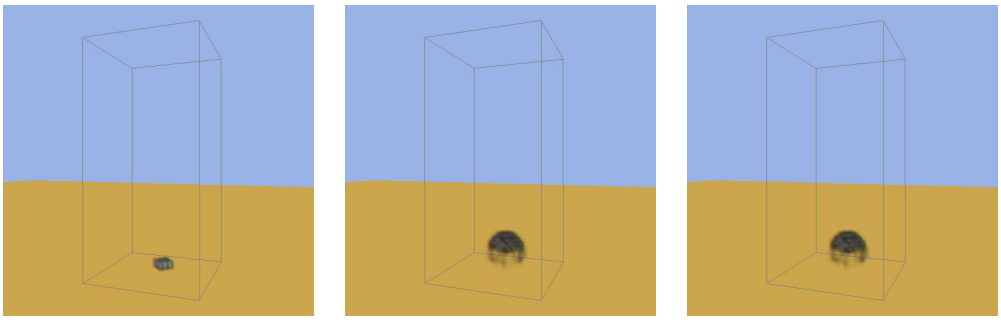


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

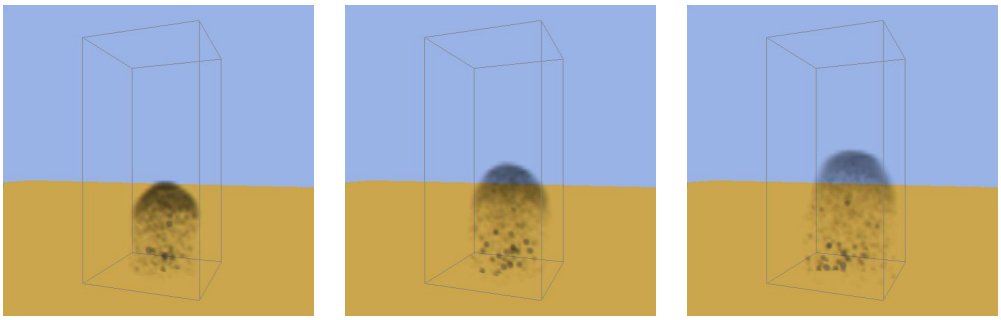


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

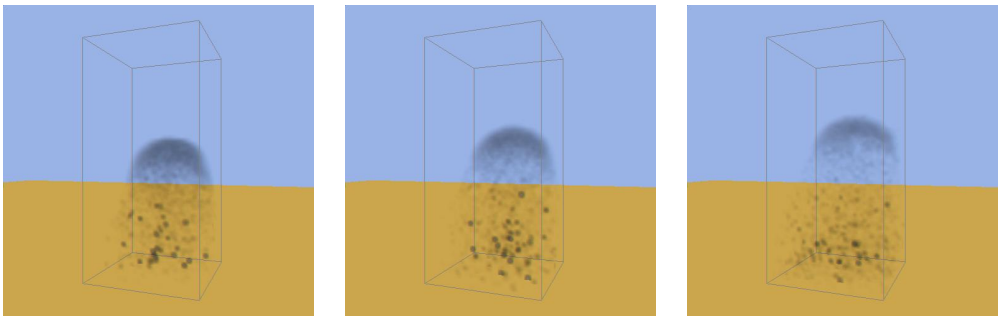
Figure 12.33: Images from test 8b.



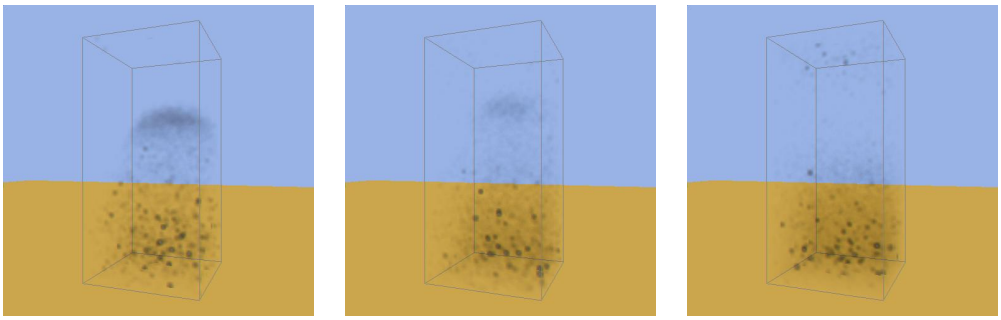
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.34: Images from test 8c.

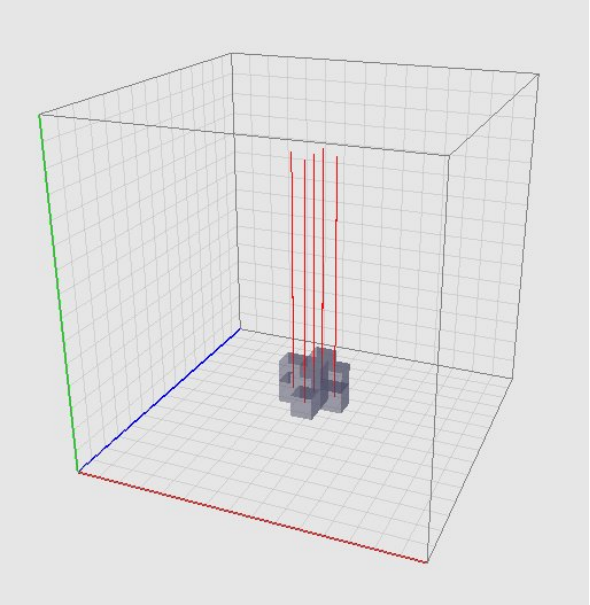


Figure 12.35: Setup for test 9a. A hot smoke source is put in the bottom of a low-resolution grid ($15 \times 15 \times 15$ cells).

Test 9a: Small Grid

When lowering the grid size an increase in frame rate is expected. In this test a grid resolution of $15 \times 15 \times 15$ is used, to get a picture of how much this increases the frame rate. This test uses the stup from figure 12.35.

From table 12.2 we see that the frame rate *is* increased, but not very much. This

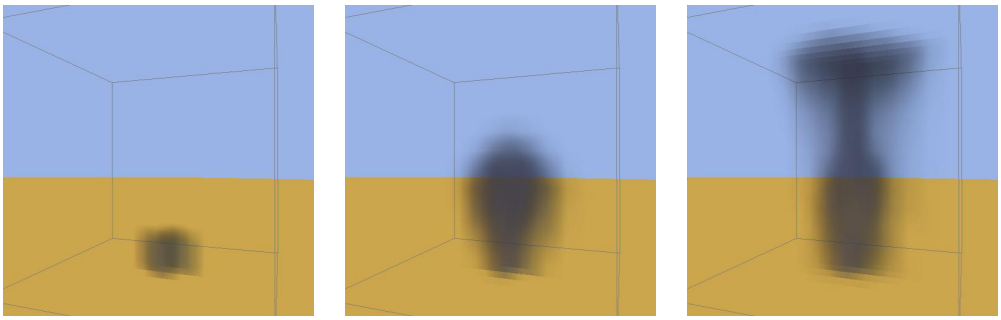
Grid Size	Running Time	Number of Frames	Avg. Frame Rate
$30 \times 60 \times 30$ (test 6b)	24.89 seconds	401	16.11 fps
$15 \times 15 \times 15$ (test 9a)	17.96 seconds	401	22.33 fps
$30 \times 30 \times 60$ (test 9b)	24.50 seconds	401	16.37 fps
$30 \times 90 \times 30$ (test 9c)	41.98 seconds	401	9.55 fps

Table 12.2: Time meassurements of the tests 9a-c.

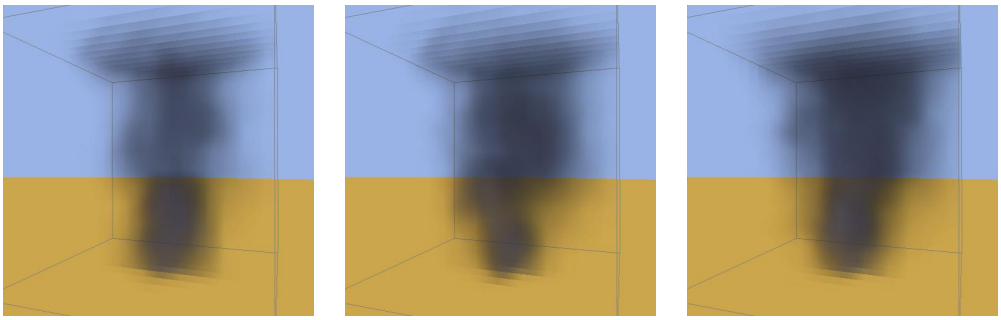
implies that the swapping between fragment shaders, switching of render contexts, and other background solver tasks are actually quite time consuming, compared to the actual solver steps. Images from this test can be seen in figure 12.36 on page 120.

Test 9b: More Tiles

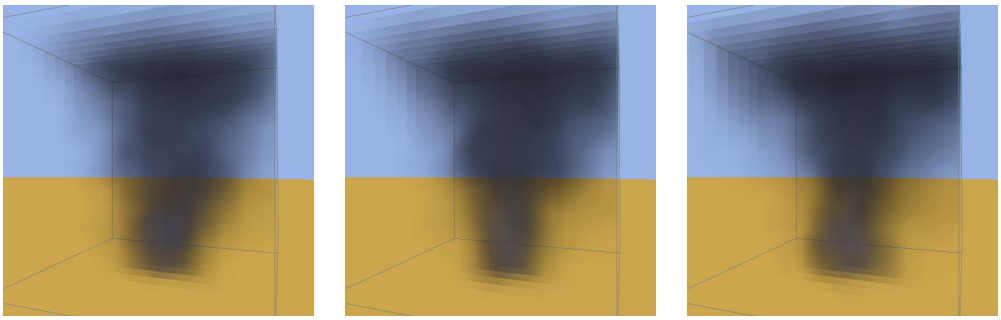
In this test a grid of size $30 \times 30 \times 60$ is used, as shown in figure 12.37 on page 121. This grid has the same number of cells as the reference grid, but a higher number of tiles in the field texture maps. This is expected to have a negative effect on the frame rate, since it



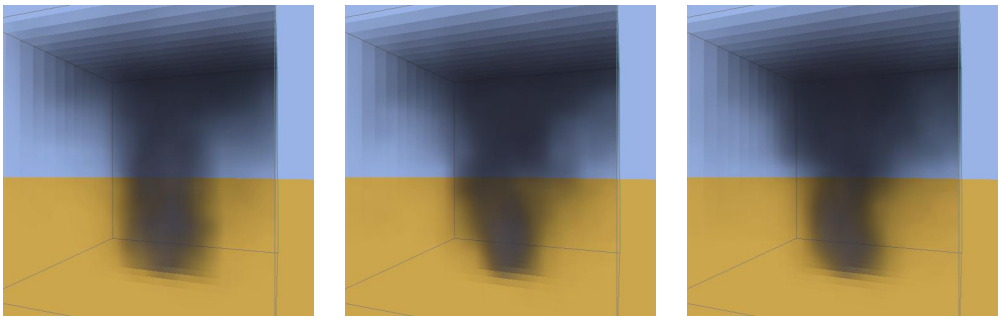
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.36: Images from test 9a.

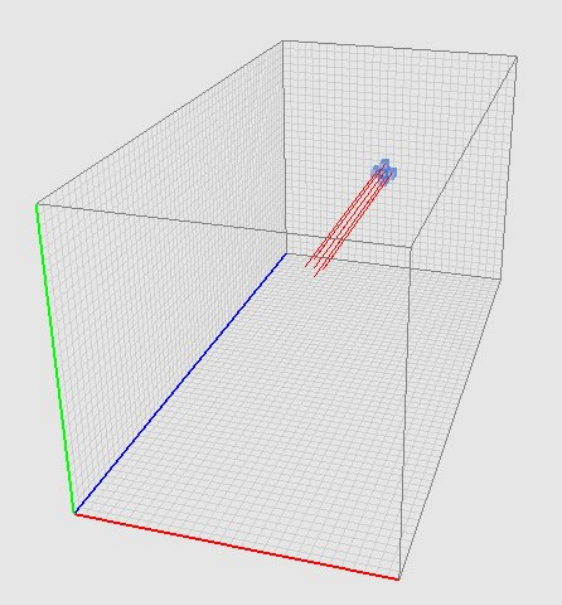


Figure 12.37: Setup for test 9b. A colored smoke source is put in the far end of a grid of size $30 \times 30 \times 60$.

requires rendering of more quads in every step of the solver. As we see in table 12.2, the frame rate is lower, but not in the extend that could be feared. The animation is shown in figure 12.38 on page 122.

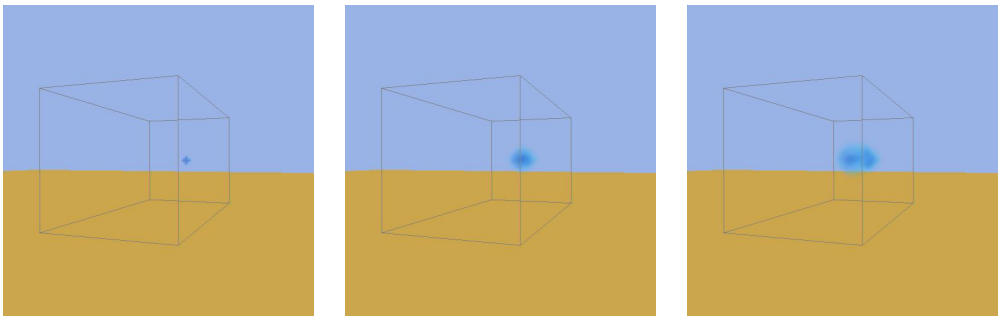
Test 9c: Large Grid

The grid size is now increased to $30 \times 90 \times 30$, thus increasing the number of cells by a factor 1.5 compared to the reference grid. The setup is shown in figure 12.39 on page 123. This should obviously lower the frame rate. As table 12.2 shows, the frame rate drops drastically, as expected. The animation of this test is shown in figure 12.40 on page 124.

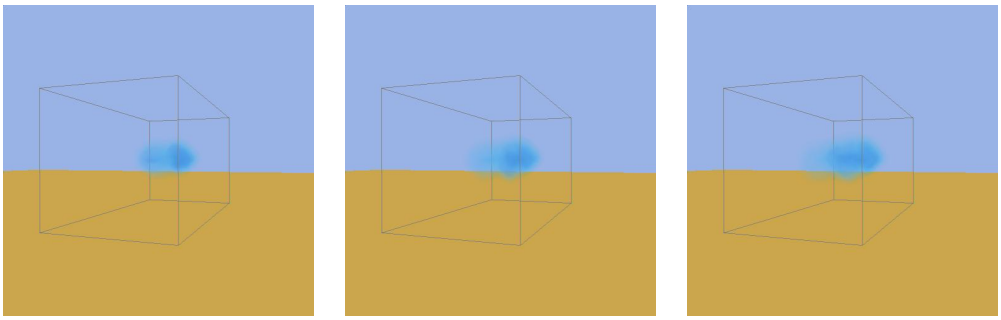
12.2.10 Test 10: Number of Iterations in the Pressure Solver

To follow up the test of the pressure solver (test 1a), this test demonstrates the effect on the visual quality and frame rate of the simulation, when changing the number of iterations used for solving the pressure in the mass-conservation step.

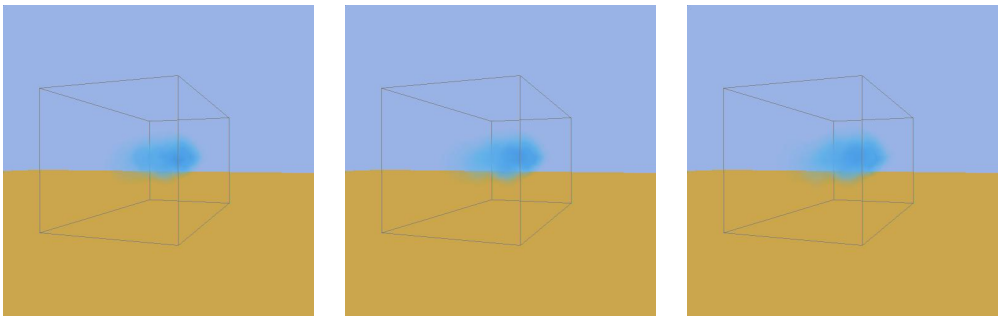
It should be clear at this point, that even though the mass-conservation step does not fully remove all divergence in a single step, animation of smoke-like behavior is still possible. In this test, we will thus only concentrate on the effects of increasing and decreasing the number of iterations. For this purpose, we use the setup from figure 12.6, on page 88 and we use the animation and time meassurements from test 6b as a refference.



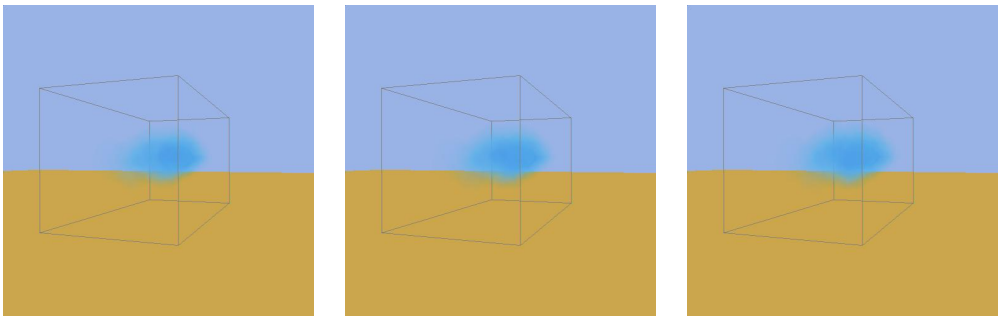
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.38: Images from test 9c.

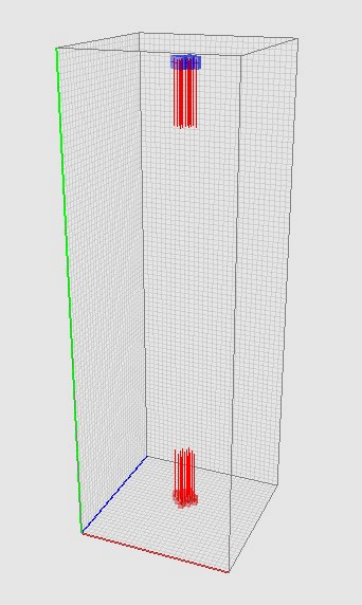


Figure 12.39: Setup for test 9c. A hot, red smoke source is put in the bottom of a high grid, and a cold, blue smoke source is put in the top. The size of the grid is $30 \times 90 \times 30$ cells.

Test 10a: 5 Iterations

Since this test uses a lot fewer iterations, the mass-conservation step is expected to be a lot quicker. From table 12.3 it shows that the frame rate is in fact much higher than in the referenced test. At the same time, we see from figure 12.41 on page 125 that

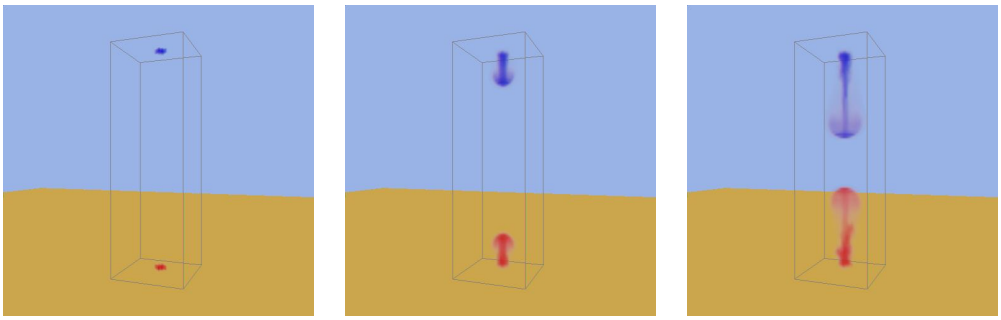
Number of iterations	Running Time	Number of Frames	Avg. Frame Rate
20 (test 6b)	24.89 seconds	401	16.11 fps
5 (test 10a)	12.18 seconds	401	32.92 fps
100 (test 10b)	88.53 seconds	401	4.53 fps

Table 12.3: Time meassurements of the tests 10a-b.

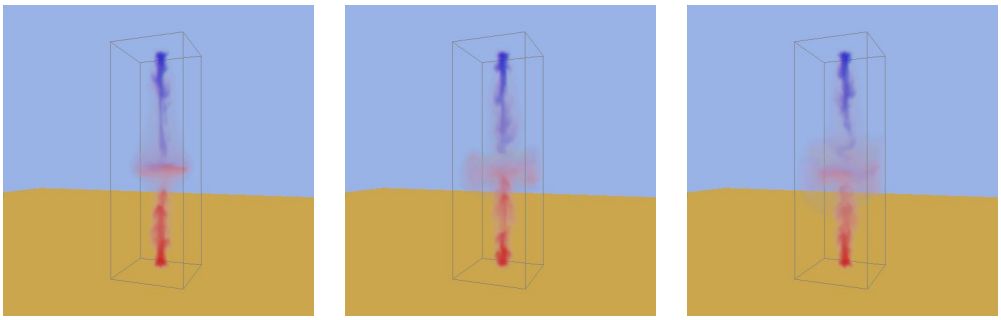
the animation still shows smoke-like behavior, although the volume-loss is a bit more expressed. However, it seems possible to lower the number of iterations for increased speed. This is a nice feature in relation to the application in a copmuter game, where the available resources depend on the exact platform running the game.

Test 10b: 100 Iterations

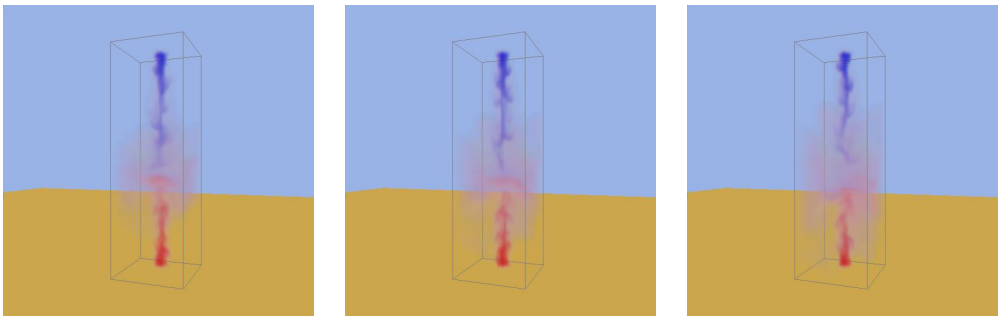
As a contrast to the previous test, this test uses more iterations, and will thus presumably run slower, but more precise. We see from table 12.3 that our presumptions are right. And as expected, figure 12.42 on page 127 shows that the animation *is* more precise; the



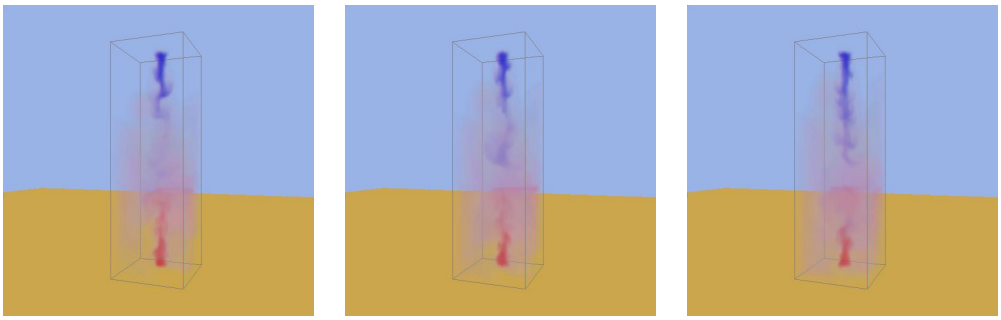
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$

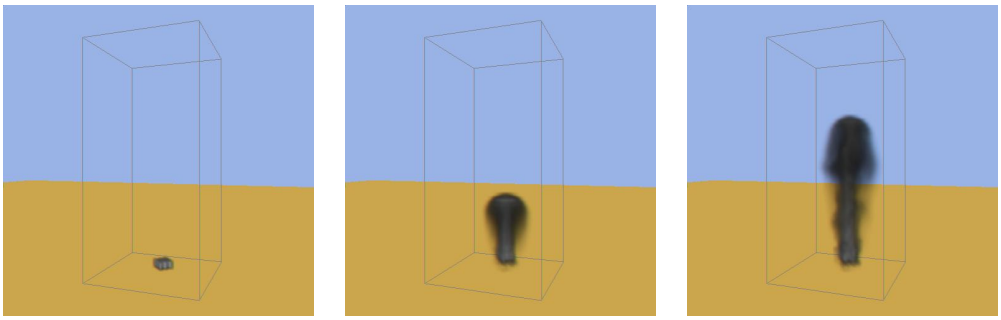


$t = 6.0s$ $t = 7.0s$ $t = 8.0s$

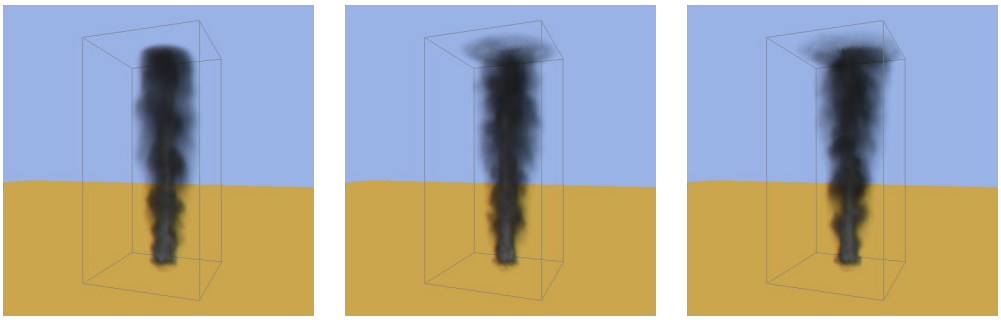


$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

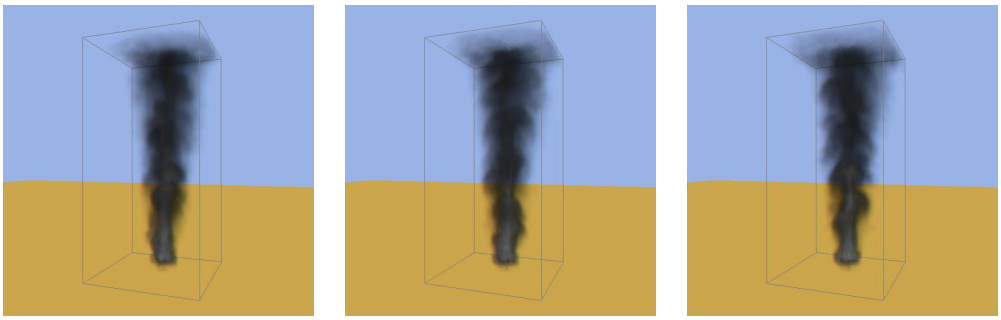
Figure 12.40: Images from test 9c.



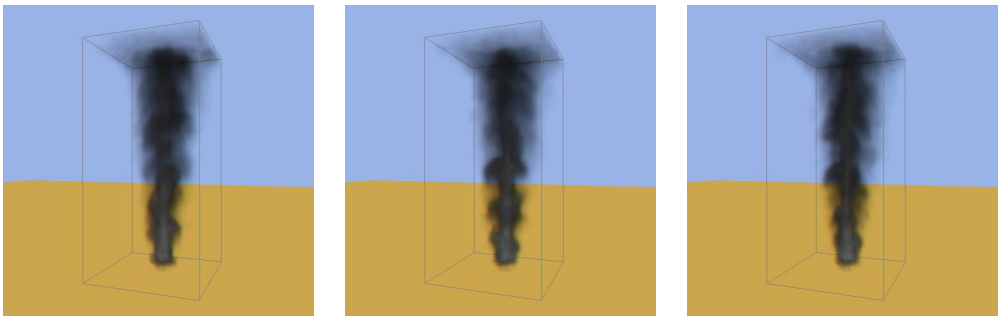
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.41: Images from test 10a.

smoke flows further down the sides of the simulatin box, indicating that the volume-loss is less expressed, compared to figure 12.23 on page 105.

With this test, we finish the test section, and move on to some general comments on the tests.

12.3 Evaluation of the Solver

We have shown that our solver can in fact produce smoke-like animations. We have presented a wide range of configuration possibilities, which have different impacts on performance and visual quality of the resulting animations.

Even though the solver is generally working as desired, there are some issues, which need to be resolved, in order for the solver to be usable in for instance a computer game.

12.3.1 Pressure Solver

One of the major issues showing up in the tests, is the imprecision of the pressure solver. After having performed the test, the code has been repeatedly searched for errors, but none have show up.

This issue does not seem to be a matter of slow convergence of the Jacobi method; increasing the number of iterations, even to the extreme, does not seem to have any major impact on this issue. Replacing the Jacobi method, with a more advanced method, is thus not expected to change this drastically.

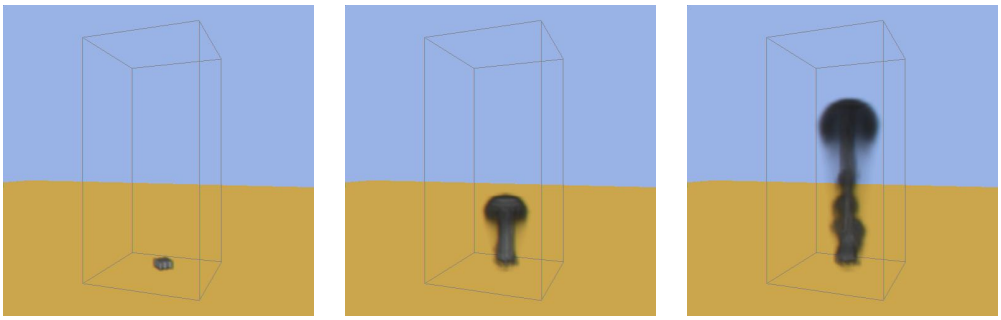
In an attempt to fix this problem, it has been tried simply to repeat the mass-conservation step a couple of times. When at the same time lowering the number of iterations per mass-conservation step to maintain frame rates, it actually seems as if higher precision, and even a higher frame rate, can be obtained. This fix, however, has some unfortunate effects on other parts of the solver, so it will only be considered as a temporary fix. The pressure solver is thus a place to work on improvements.

12.3.2 Advection

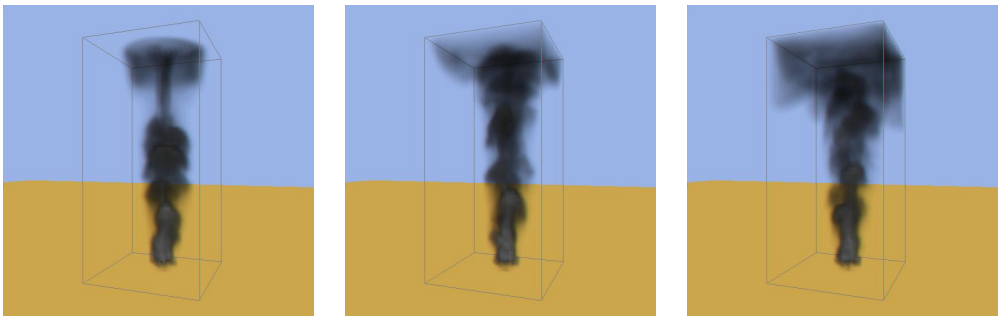
In test 2a, we experienced a slight imprecision in the advection of the velocity field. After searching the advection fragment shader repeatedly for this error we still cannot find any errors in the fragment shader. As we have seen in the rest of the tests, this imprecision is barely visible, since it is probably damped by addition of the rest of the solver steps. We do, however, that in some of the tests the smoke has a slight tendency to move more in the positive directions than in the negative, so this is also an issue that should be worked on.

12.3.3 Realism

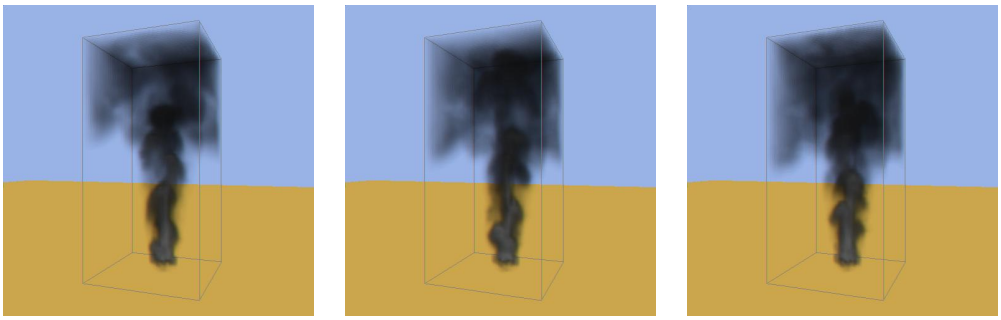
The test setups presented in this section are all very simple, and can thus not be directly compared to a real-life setup. Intuitively, however, the animations of many of the tests do appear very smoke-like.



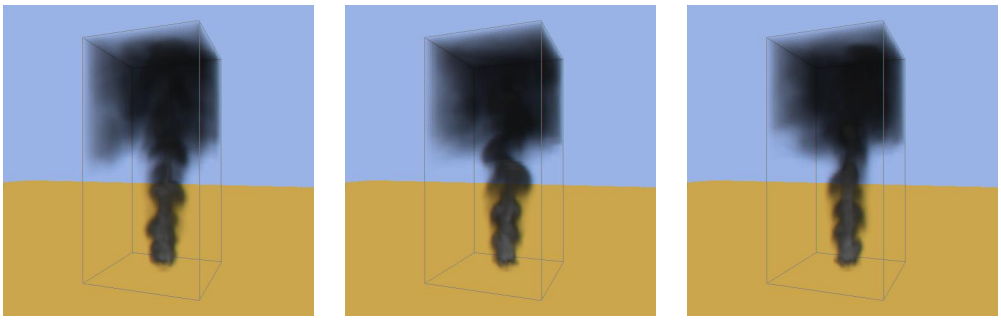
$t = 0.0s$ $t = 1.0s$ $t = 2.0s$



$t = 3.0s$ $t = 4.0s$ $t = 5.0s$



$t = 6.0s$ $t = 7.0s$ $t = 8.0s$



$t = 9.0s$ $t = 10.0s$ $t = 11.0s$

Figure 12.42: Images from test 10b.

The inclusion of thermal buoyancy and vorticity confinement adds nice looking features to the smoke, and the simple blending of the density field tiles is considered very effictive for the visualization of a diffusive substance, such as smoke.

On the negative side, the volume-loss – both that related to the imprecision of the pressure solver, and that related to the semi-Lagrangian advection scheme – does disturb the realism, but many effects can still be simulated rather realistically.

Comparred to previously observed computer game effects, it is our oppinion that effects, with the realism presented here, will actually improve game playing experiences.

12.3.4 Simulation Speed

In tables 12.1, 12.2, and 12.3 we see frame rates in the range 4 to 32. In most cases, however, the frame rate is about 15 to 20.

To perform real-time simulation at a frame rate of 15 frames per second, the time step has to be $dt > 1/15 = 0.0666667$. This is possible, but as we have seen in test 8b, a time step of $dt = 0.1$, corresponding to 10 fps, visual quality is disturbed. Thus, at a frame rate of 15 fps, simulation at a decent quality in real-time is barely possible. Taking into account that this is when using all power of the GPU, this is not even close to be usable for implementation into a computer game.

Although we have no concrete reference, we do however presume a great increase in frame rates, compared to a regular CPU implementation of fluid simulation – we estimate this to a factor 5-10, depending on the system configuration.

Lowering the number of iterations increases the frame rate dramatically. If the simulation is allowed to run with only 5 iterations, a frame rate of about 30 fps can be obtained. At this rate, real-time is possible.

The speed of the simulation is very dependent of the hardware. All frame rates mentioned here are based on the hardware used in the tests, an ATI Radeon 9600 PRO. Since the evolution of the power of graphics harware is even faster than seen on CPUs, at the end of this project, this graphics card will already be considered outdated. Furthermore, this graphics card is in the low-end market. A significant increase in performance is expected, just by using a newer and more expensive graphics card.

12.3.5 Interaction with Objects

The handling of internal boundaries is very simple, and thus crude results are obtained. Handling internal boundaries differently, and more correctly, would require changes at the cost of speed. To fully correct the handling of internal boundaries, it is our conviction that a change to a staggered representation would in fact be better, since there will always be unsolvable issues with obstacles on a collocated grid.

We do, however, get a somewhat usable result, when the objects are solid and cover at least a couple of cells in each direction. Thus, for the purpose of a computer game enging, we presume that our implementation of internal boundaries can be used.

Chapter 13

Conclusion

We have described a way to simulate fluid using programmable graphics hardware, and the method has been implemented in a demo application for the specific case of animating smoke. Despite some inaccuracies, in the implemented methods as well as in the general computation on graphics hardware, this demo application is able to produce realistically looking smoke animations at frame rates allowing real-time simulation, which was our primary goal.

As expected, the simulation still does not run fast enough for the use in a computer game engine. We prove, however, that the simulation of advanced fluid effects in computer games is just around the corner.

13.1 Future Work

During the work with this project, a lot of ideas for extensions and related projects have sprung to our mind. In this final section, we will list some of these ideas, as encouragement for future work in this filed.

Improved Simulation Speed

First of all, making the solver running in sufficient speed for computer game implementation would be an interesting area of focus. The field of computer game simulation is widely renowned for the tricks and hacks applied to effects, to achieve what at first encounter seems impossible. We expect this to be the case of fluid effects as well. Experienced game programmers might see possible simplifications and optimizations, which we have not noticed.

Better Interaction with Objects

We have experimented with a simple method for simulating interaction between the fluid and stationary objects. This method has proven to be uselessly insufficient in the general case, and merely acceptable for specially designed objects. Certainly, for the use in a computer game engine, an for the sake of simulating more advanced fluid behavior in general, a better method for interaction with stationary object should be implemented.

As part of this, interaction between the fluid and *moving* objects could be considered. This brings us on to the next idea.

Staggered Grids

We have experienced slight imprecisions in some of the solver steps. Since some of these may be numerical dissipation caused by the collocated representation, it would be interesting to implement a similar solver on a staggered representation. This may also improve the precision of interaction with objects.

Improved Visual Presentation

The visual quality has never been a highly prioritized part of this project. For the animations to be usable, even for the use in computer games, higher visual quality is required. We have already given some ideas on this, in chapter 10, but many other methods exist.

Other Types of Fluids

Smoke is one of the simplest of the different types of fluids mentioned in chapter 3. Using the method of this project for simulating other types of fluid phenomena is an obvious extension. In [17], a similar method has already been used for advanced cloud simulation and, in [21], the simulation of explosions with a similar method has also been tried successfully. Other types, such as liquids, could be considered.

Non-Cartesian Non-Uniform Grids

For the use in field of computer games, where the voxelization of the environment is generally undesired, other possible representations of the fluid volume could be considered.

In [28], Rasmussen et al. produce very high-quality animations of large-scale fluid phenomena by extrapolating a 3D simulated velocity field from a 2D simulation. Following this idea, advanced techniques such as NURB interpolation between tiles could extend the usabilities of the flat 3D texture almost infinitly.

Bibliography

[1] OpenGL Architectural Review Board (ARB). *The OpenGL Reference Manual – The Bluebook*. Addison-Wesley Publishing Company, Inc., 2002. Available at `http://www.opengl.org/documentation/blue_book_1.0/`.

[2] Ati corporation homepage. `http://www.ati.com`.

[3] Smartshader(tm) technology white paper. Available at `http://www.ati.com/products/pdf/smartshader.pdf`.

[4] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG)*, 22(3):917–924, 2003.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[6] James W. Demmel. *Applied Numerical Linear Algebra*, chapter 6, pages 265–361. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[7] Microsoft directx homepage. `http://www.microsoft.com/windows/directx/`.

[8] Jens Egeblad, Michael Gram Haagensen, and Marinus Rørbech. En adaptiv model til animering af væsker. Technical report, Department of Computer Science, Copenhagen University, 2002. Available at `ftp://ftp.diku.dk/diku/image/Roerbech021104.pdf`.

[9] J. Douglas Faires and Richard L. Burden. *Numerical Analysis*. Brooks/Cole Publishing Company, 6th edition, 1997.

[10] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM Press, 2001.

[11] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley Publishing Company, Inc., 2nd edition, 1996.

[12] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30. ACM Press, 2001.

[13] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical models and image processing: GMIP*, 58(5):471–483, 1996.

[14] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. *Computer Graphics*, 31(Annual Conference Series):181–188, 1997.

[15] General purpose gpu programming homepage. `http://www.gpgpu.org`.

[16] Stefan Guthe, Stefan Gumhold, and Wolfgang Straßer. Texture particles: Interactive visualization of volumetric vector fields.

[17] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101. Eurographics Association, 2003.

[18] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language*, 2003. Available at `http://www.opengl.org/documentation/oglsl.html`.

[19] David Kirk. *CG Toolkit User's Manual*, 2003. Available at `http://developer.nvidia.com/cg`.

[20] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[21] Jesper Molbech Taxbøl. Real-time explosions based on fuid dynamics. Master's thesis, Technical University of Denmark (DTU), 2004.

[22] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[23] Michael Meissner and Stefan Guthe. Interactive lighting models and pre-integration for volume rendering on pc graphics accelerators. In *Graphics Interface*. Canadian Information Processing Society, 2002.

[24] Michael Meissner, Stefan Guthe, and Wolfgang Straßer. Higher quality volume rendering on pc graphics hardware. Technical Report WSI-2001-12, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, April 2001.

[25] Nasa geometry lab image gallery. `http://geolab.larc.nasa.gov/cgi-bin/Bene/Gallery/Gallery.pl`.

[26] Nvidia corporation homepage. `http://www.nvidia.com`.

[27] Opengl extension registry. `http://oss.sgi.com/projects/ogl-sample/registry`.

[28] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. *ACM Trans. Graph.*, 22(3):703–707, 2003.

[29] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.5)*, 1992–2002. Available at `http://www.opengl.org/documentation/spec.html`.

[30] S. K. Som and G. Biswas. *Introduction of Fluid Mechanics and Fluid Machines*. Tata McGraw-Hill Publishing Company, Lmt., New Delhi, India, 2002.

[31] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[32] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, 2003. Available from `http://www.dgp.utoronto.ca/~stam/reality/Research/pub.html`.

[33] John Steinhoff and David Underhill. Modification of the euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluid*, 6(8):2738–2744, 1994.

[34] D.J. Tritton. *Physical Fluid Dynamics*. Clarendon Press, Oxford, UK, 2nd edition, 1988.

Appendix A

Discrete Differentiation

In order to process the equations of fluid motion with a computer, the equations need to be discretized. For this discretization, one of the main tasks is to find out how to compute the derivative f' and the second derivative f'' of some function $f : \mathbb{R} \to \mathbb{R}$.

A.1 Finite Differencing

For the purpose of this report, it is of great importance that the derivatives can be computed as fast as possible, rather than precise. Thus, the simple method of first order finite differencing, as used in [10, 13, 14, 32], seems appropriate.

By using *Taylor's Theorem* (see for instance page 10 in [9]) the derivative f' can be written in the following three ways

$$f'(x_0) = \frac{1}{h}\Big(f(x_0+h) - f(x_0)\Big) - \frac{h}{2}f''(\xi_1), \tag{A.1}$$

$$f'(x_0) = \frac{1}{h}\Big(f(x_0) - f(x_0-h)\Big) - \frac{h}{2}f''(\xi_2), \tag{A.2}$$

$$f'(x_0) = \frac{1}{2h}\Big(f(x_0+h) - f(x_0-h)\Big) - \frac{h^2}{6}f^{(3)}(\xi_3), \tag{A.3}$$

where $h \in \mathbb{R}$, $\xi_1 \in [x_0, x_0+h]$, $\xi_2 \in [x_0-h, x_0]$, and $\xi_3 \in [x_0-h, x_0+h]$. By discarding the second- and higher-order terms, the following approximations to the derivatives are formed

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}, \tag{A.4}$$

$$f'(x_0) \approx \frac{f(x_0) - f(x_0-h)}{h}, \tag{A.5}$$

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0-h)}{2h}. \tag{A.6}$$

The approximations (A.4), (A.5), and (A.6) are known as *forward differencing*, *backward differencing*, and *central differencing*, respectively.

Similarly, the second derivative of f can be written as

$$f''(x_0) = \frac{1}{h^2}\Big(f(x_0+h) - 2f(x_0) + f(x_0-h)\Big) - \frac{h^2}{12}f^{(4)}(\xi), \tag{A.7}$$

where $\xi \in [x_0 - h, x_0 + h]$, yielding the approximation

$$f''(x_0) \approx \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2}. \tag{A.8}$$

These approximations can also be used for vector fields, by applying them componentwise. E.g. we have

$$\frac{\partial v^x(x_0, y_0, z_0)}{\partial x} \approx \frac{v^x\left(x_0+h, y_0, z_0\right) - v^x\left(x_0-h, y_0, z_0\right)}{2h}. \tag{A.9}$$

Appendix B

Solving Linear Systems Iteratively

Since the implementation of a solver of linear systems is an important part of getting a fluid solver up and running, this appendix will give a short introduction to iterative methods, and the Jacobi iterative method, which is used in this work, is introduced. This introduction is based on section 7.3 in [9]. Other iterative techniques, such as the Conjugate Gradient method, can be found in [6].

B.1 Linear Systems and Iterative Techniques

We wish to solve the linear system

$$A\mathbf{x} = \mathbf{b}, \tag{B.1}$$

where the matrix A and the vector $\mathbf{b}$ define the LHS and RHS, respectively, of the linear equations written in matrix notation, and $\mathbf{x}$ is the unknown vector we are looking for.

The idea of an iterative approach is to start with an initial guess $\mathbf{x}^{(0)}$, and use this to generate a sequence of vectors $\{\mathbf{x}^{(m)}\}_{m=0}^{\infty}$ converging to $\mathbf{x}$.

By transforming equation (B.1) into an *equivalent* system on the form

$$\mathbf{x} = T\mathbf{x} + \mathbf{c}, \tag{B.2}$$

it follows from the *Fixed-Point Theorem* (see for instance [9] page 61), that the sequence $\{\mathbf{x}^{(m)}\}_{m=0}^{\infty}$ can be computed by

$$\mathbf{x}^{(m)} = T\mathbf{x}^{(m-1)} + \mathbf{c}, \tag{B.3}$$

provided we choose the proper matrix T and vector $\mathbf{c}$.

Depending of the choice of T and $\mathbf{c}$, different convergence *speeds* (in term of numbers of iterations) can be obtained. Normally, the fastest convergence is desired, thus requiring a low number of iterations to produce an acceptable approximation, however, the computational cost of each iteration should also be considered.

B.2 The Jacobi Iterative Method

We now explain how a linear system on the form (B.1) can be transformed into an equivalent system, on the form (B.2), in a way that allows easy computation of each iteration.

By splitting the matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{12} & \cdots & a_{nn} \end{pmatrix}$$

into its diagonal part

$$D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

and strictly upper and lower parts

$$-U = \begin{pmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & -a_{n-1,n} \\ 0 & \cdots & \cdots & 0 \end{pmatrix} \quad \text{and} \quad -L = \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ -a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & \cdots & -a_{n,n-1} & 0 \end{pmatrix}$$

we can reformulate equation (B.1) as

$$(D - U - L)\mathbf{x} = \mathbf{b}. \tag{B.4}$$

From this follows

$$\begin{aligned} & D\mathbf{x} = (U + L)\mathbf{x} + \mathbf{b} \\ \iff \quad & \mathbf{x} = D^{-1}(U + L)\mathbf{x} + D^{-1}\mathbf{b}. \end{aligned} \tag{B.5}$$

We can thus compute the sequence of approximations as

$$\mathbf{x}^{(m)} = T_j\mathbf{x}^{(m-1)} + \mathbf{c}_j, \tag{B.6}$$

where $T_j = D^{-1}(U + L)$ and $\mathbf{c}_j = D^{-1}\mathbf{b}$.

B.3 Application To a Simulation Grid

The linear systems needed to be solved in the context of this report are formed on the basis of the relations between a cell $c_{i,j,k} \in \mathcal{G}$ in the simulation grid $\mathcal{G}$ and its left, right, lower, upper, front, and back neighbors,

$$c_{i-1,j,k},\ c_{i+1,j,k},\ c_{i,j-1,k},\ c_{i,j+1,k},\ c_{i,j,k-1},\ c_{i-1,j,k+1} \in \mathcal{G}.$$

Thus, the matrix A will be very sparse and formatted consistenly for each cell.

Each cell in the simulation grid gives rise to a row in the matrix. This row describes the relations between this cell and all other cells in the grid, thus, each cell is also represented

by a column in the matrix. The matrix, hence, has dimensions $(W \cdot H \cdot D) \times (W \cdot H \cdot D)$, where W, H, and D are the width, height, and depth of the grid $\mathcal{G}$, respectively, and the vector $\mathbf{x}$ has dimension $(W \cdot H \cdot D)$, holding one value per cell in the grid.

Since a cell only relates to it self and its six neighbours, only seven elements in each row will be non-zero. By numbering the elemens of the vectors and the matrix using the indices of the corresponding cells, we can thus express equation (B.6) for a single cell in the grid in the form

$$\begin{aligned} x_{i,j,k}^{(m)} \quad = \quad & \frac{a_{i-1,j,k}x_{i-1,j,k}^{(m-1)} + a_{i+1,j,k}x_{i+1,j,k}^{(m-1)}}{a_{i,j,k}} + \\ & \frac{a_{i,j-1,k}x_{i,j-1,k}^{(m-1)} + a_{i,j+1,k}x_{i,j+1,k}^{(m-1)}}{a_{i,j,k}} + \\ & \frac{a_{i,j,k-1}x_{i,j,k-1}^{(m-1)} + a_{i,j,k+1}x_{i,j,k+1}^{(m-1)}}{a_{i,j,k}} - \frac{b_{i,j,k}}{a_{i,j,k}}. \end{aligned} \tag{B.7}$$

An approximation vector $\mathbf{x}^{(m)}$ can thus be computed by computing the elements one at a time. Since the pattern of elements used from A is similar for each cell, the job of generating the matrix can be spared. Usually, all six elements relating to neighbor cells are even equal, simplifying equation (B.7) even further.