Eulerian Smoke Simulation on the GPU
MSc Computer Animation and Visual Effects
Master Thesis

Nikolaos Verigakis

N.C.C.A Bournemouth University
August 19, 2011

Contents

List of Figures

Abstract

The realistic simulation of smoke motion has always been a popular demand in the visual effects industry. However, systems that implemented this fluid behavior have been predominately non-interactive, high-quality offline rendering systems. The purpose of this thesis has been to provide an efficient and interactive tool for the realistic simulation of smoke. A fast and efficient method for real-time Eulerian smoke simulation on the GPU is presented. The implemented application uses an OpenCL gas solver, along with a real-time iso-surface extraction method for the rendering of the fluid.

Chapter 1

Introduction

Realistic smoke effects have always been a popular demand in the visual effects industry (Stam, 1999). Being a moving unit by definition, smoke enriches any scenery, even a standstill landscape. The more realistically smoke is rendered, the more sight-attracting it becomes. Therefore, it is very important to render smoke as carefully and meticulously as it could be, in order to get its full value in the visual scenery.

The motion of smoke can be observed in many places in everyday life (e.g., the billowing smoke of a cigarette, or the fumes from an exhaustion pipe) and hence, the viewers have certain expectations of what they see in a movie. Smoke is a type of fluid, and the realistic simulation of fluid motion has started to interest the computer graphics community since the 1980's. However, the study of fluids and fluid dynamics has a much longer history (Griebel et al., 1997).

1.1 Computer animation system requirements

In computer animation applications, the appearance and the motion of the fluids are of great importance; physical accuracy plays a subsidiary role which sometimes is even irrelevant. Moreover, it is essential to the animator who is using the application to get interactive feedback and control on the simulation. In other words, it is important to the user to have the ability to tamper with the simulation, despite physical inaccuracies that may occur. Real-time results can significantly enhance an animation production pipeline as it speeds-up the look-development of the fluid effect.

1.2 Goal and objectives

The purpose of this thesis is to try to provide computer animators with a reliable and easy to handle tool, which will enable them to create realistic scenes with the use of smoke simulation. The basic requirements of the application are the following:

- Interactive feedback.
- Control over the simulation.
- High resolution simulations.

Chapter 2

Fluid simulation

This chapter serves as a brief introduction to fluid dynamics and fluid simulation. Firstly, fluid flow is defined along with a suitable mathematical model. Subsequently, some fundamental concepts of numerical simulation will be established, in order to finally construct a practical framework for the simulation of smoke.

2.1 Fluid motion

2.1.1 Fluids

Fluids, by definition are substances which cannot resist shear stress when at rest (Griebel et al., 1997). They pose little resistance to deformation and are characterized by their ability to take the shape of their container. Fluids can be categorized in two basic types: gases (e.g., smoke, air) and liquids (e.g., water, oil)[1]. They can also be distinguished as either compressible or incompressible, in respect to whether their volume remains constant over time. However, in computer animation it is very common to assume an incompressible and homogeneous fluid (Harris, 2004). The incompressibility condition implies that the fluid's volume doesn't change or, more specifically, that the volume of any sub-region remains constant over time. Additionally, the homogeneity of the fluid denotes that its density is constant in space. As a result, the density of the fluid remains constant in both time and space. It must be noted, however, that this simplifying assumption does not constrain the realistic simulation of any type of fluid.

2.1.2 Fluid flow

Fluid flow describes the motion of fluids. This motion is created by both interactions between fluid particles and by those between the fluid and solid objects

[1] Plasmas and plastic solids are also fluids; however, they are of a lesser importance in the field of computer animation.

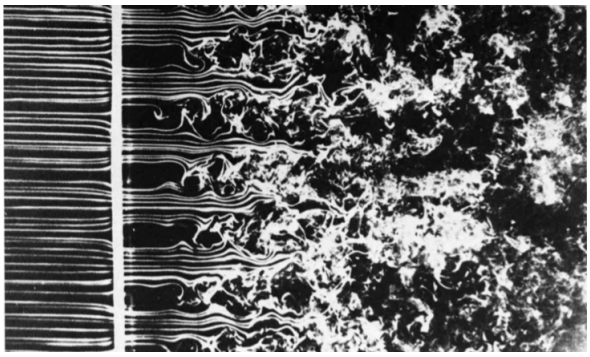

Figure 2.1: A uniform laminar stream of smoke passing through a perforated plate. Instability of the shear layers leads to turbulent flow downstream. Photograph taken from (Van Dyke, 1988).

(Griebel et al., 1997). Fluid flow is determined by two elementary properties: viscosity and inertia. Viscosity describes a set of frictional forces that push the fluid towards a resting condition, while inertia represents a fictitious force that is proportional to the mass acted upon (it is thus proportional to the density of the fluid).

In order to understand the interaction between these two elementary properties laminar flow can be examined, since it is the simplest type of flow [2]. In laminar flow the fluid can be conceptualized as a set of individual layers (e.g., a deck of playing cards); when these layers are set in uniform motion and the forward motion on the bottom layers stops, then the layers above will continue to slide due to their *inertial* force. Additionally, the force on the bottom layers transmits to other layers through friction (*viscosity*).

In turbulent flows on the other hand, due to small friction particles belonging to different layers are seen to mix. This effect leads to an increased effective viscosity called "turbulent eddy viscosity". Figure 2.1 shows the transition between a laminar and a turbulent flow.

2.2 Mathematical description of flow

Fluid flows in computer animation can be described mathematically by a set of partial differential equations: the incompressible Navier-Stokes equations (Griebel et al., 1997). These equations describe the interdependence between velocity and pressure in space and time (Bridson, 2007).

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{f} + \nu \nabla \cdot \nabla \vec{u} \tag{2.1}$$

[2] From the latin word lamina which means thin sheet.

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

Equation 2.1 is the momentum equation. It actually represents Newton's second law, according to which a body of mass subject to a net force undergoes acceleration ($\vec{F} = m\vec{a}$). In simple terms, this equation describes the acceleration of the fluid due to forces acting upon it. The momentum equation stands actually for 3 equations in a wrapped up form, because the fluid's velocity $\vec{u}$ is a vector quantity. With some minor rearrangements, equation 2.1 can be rewritten as follows:

$$\frac{\partial u}{\partial t} = -\vec{u} \cdot \nabla u - \frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla u + f_x$$

$$\frac{\partial v}{\partial t} = -\vec{u} \cdot \nabla v - \frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla v + f_y$$

$$\frac{\partial w}{\partial t} = -\vec{u} \cdot \nabla w - \frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla w + f_z$$

Equation 2.2 is the continuity equation. This equation enforces the previously mentioned incompressibility condition. Thus, the momentum and continuity equations describe the motion of an incompressible and homogeneous fluid.

The incompressible Navier-Stokes equations at first sight may appear complicated. However, when the role of all individual terms in the formulas becomes clear then the actual meaning of these equations is fully revealed and it can be easily comprehended. In the following sub-section the equation terms will be dissected into their elementary components.

2.2.1 Terms in the Navier-Stokes equations

The most significant quantity which characterizes the state of the fluid at any specific time is its velocity $\vec{u}$ (Harris, 2004). The velocity of the fluid is represented as a vector field $\vec{u} : R^3 \rightarrow R^3$. A vector field is a map that assigns a vector-valued function $\vec{u}(x)$ for every position $x = (x, y, z)$, in a subset of a Cartesian space. A vector field can be visually represented by assign each point in the field an arrow pointing to a vector direction (e.g., see figure 2.2).

The symbol p represents the pressure of the fluid and signifies the force per unit area that the fluid exercises on its surroundings (and itself) (Bridson, 2008). Pressure builds up in the fluid when force is applied, as fluid molecules in the vicinity of the force push on those that lie farther away (Harris, 2004). The

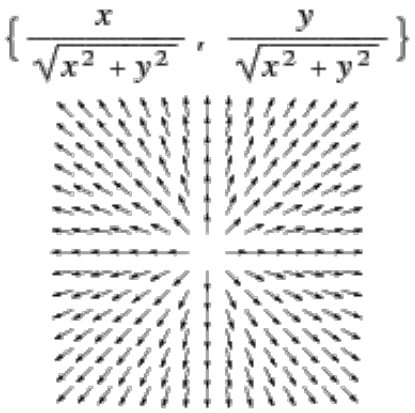

Figure 2.2: A vector field plot. Figure modified from (Weisstein, 2011).

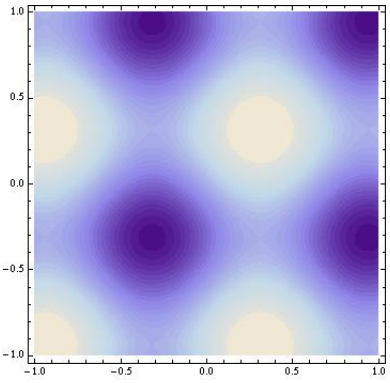

Figure 2.3: A visual representation of a scalar field. Figure taken from (Biały, 2010).

pressure of the fluid is a scalar field; i.e., a map $p : R^3 \rightarrow R$ that assigns each point x in the Cartesian space a scalar-valued function $p(x)$. A scalar field can be visually represented by mapping the intensity of the field to a colour value (e.g., see figure 2.3).

The greek letter ρ in equation 2.1 represents the constant density of the fluid. For water, ρ is approximately $1000\,kg/m^3$, and for air it is about $1.3\,kg/m^3$ (Bridson, 2008). The greek letter ν in the momentum equation represents the fluid's kinematic viscosity (a viscous acceleration). Finally, the vector quantity $\vec{f}$ represents any external forces that act upon the fluid.

2.2.1.1 Differential operators

The Nabla (or del) operator ∇ in equations 2.1 and 2.2 has three different applications: the gradient, the divergence and the Laplacian operators (Harris, 2004).

The gradient of a scalar field is a vector of partial derivatives of the scalar field (Bridson, 2008) (see equation 2.3). The gradient points in the direction of steepest descent, so the negative pressure gradient points away from high-pressure regions, towards low-pressure regions. Thus, the pressure gradient represents the imbalance in pressure or how high pressure regions push on lower-pressure regions.

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

The divergence operator is the sum of partial derivatives of a vector field (it can only be applied to vector fields) and its output is a scalar quantity (see equation 2.4). Divergence's physical significance is the rate at which "density" exits a given region of space. Thus, in the absence of creation or destruction of matter, the density within a region of space can change only by flowing into or out of that region.

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{2.4}$$

The Laplacian is the divergence of the gradient. It measures how far lies a quantity from the average around it. As a result, the Laplacian of the velocity field once integrated over the fluid volume provides a viscous force. In vector fields the Laplacian is applied on every component separately (see equations 2.5 to 2.7).

$$\nabla \cdot \nabla u = \nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \tag{2.5}$$

$$\nabla \cdot \nabla v = \nabla^2 v = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \tag{2.6}$$

$$\nabla \cdot \nabla w = \nabla^2 w = \frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \tag{2.7}$$

2.2.2 The Euler equations

In the turbulent flow of smoke, viscosity does not play a significant role, that is why it is possible to drop it completely. The incompressible Navier-Stokes equations without the viscosity term are called *the incompressible Euler equations* and they describe incompressible, inviscid fluids (Bridson, 2008) (see equations 2.8 and 2.9).

$$\frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla p + \vec{f} \tag{2.8}$$

$$\nabla \cdot \vec{u} = 0 \tag{2.9}$$

From now on and until the conclusion of this thesis the Euler equations will be assumed as the employed mathematical model of fluid flow.

2.3 Solving the Euler equations

The Euler equations cannot be solved analytically for every physical configuration (Harris, 2004). However, numerical integration techniques can be used to solve these equations incrementally, by discretizing the fluid domain.

There are two distinct approaches to discretize a physics problem in order to numerically simulate it: the Lagrangian and the Eulerian approach (Wicke et al., 2007). Lagrangian methods discretize the material, and thus the simulation elements move with it. In fluid simulation particle based discretizations are used, so that each particle has a position $\vec{x}$ and a velocity $\vec{u}$. Each particle of the discretization represents a part of the fluid, so mass loss due to numerical error is not an issue. Moreover, moving boundaries and free surfaces are easy to control, since the discretization moves along with the fluid. However, Lagrangian methods are not accurate in dealing with spatial derivatives on an unstructured particle cloud (Bridson, 2008). A popular Lagrangian method in computer graphics is the Smooth Particle Hydrodynamics method.

Eulerian methods discretize the space in which the fluid moves and thus the fluid moves through the simulation elements, which are fixed in space. The simulation domain is divided into a set of volume elements called voxels. The discretization of space does not depend on the fluid and for this reason large deformations such as those occurring in fluid motion can be managed efficiently. Eulerian methods are more robust in numerically approximating the spatial derivatives of the Euler equations. However, since the discretization of the simulation domain does not change with the fluid's shape, interface tracking and moving boundaries can be troublesome. Also, mass loss is possible to occur, due to numerical dissipation and as a result it may lead to an incorrect viscous force. Figure 2.4 shows a comparison between the Eulerian and the Lagrangian methods.

Hybrid methods that combine the two approaches also exist. In such methods, particles can be advected through the velocity field and measurements can be made on each particle. For example, smoke concentration can be measured for each particle in the simulation.

In this thesis the Eulerian approach has been selected for the discretization of the fluid domain, due to the numerical accuracy that it offers.

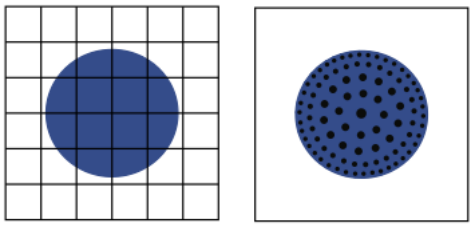

Figure 2.4: A sphere of material discretized using Eulerian and Lagrangian methods. Figure modified from (Wicke et al., 2007).

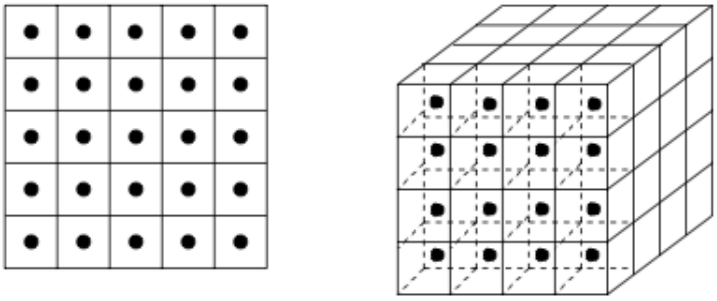

Figure 2.5: A cell-centered grid. Figure taken from (Stam, 1999).

2.3.1 Discretizing the fluid

In the Euler discretization, the vector and scalar-valued functions must be mapped onto a parameterized space, usually a Cartesian grid (Stam, 1999). The simplest structure that can be employed is a collocated (cell-centered) grid. In such a discretization, both vector and scalar fields are sampled at the center of each cell (see figure 2.5).

Using this method all the differential operations can be approximated using finite differences. The central differences for each one of the differential operators is described bellow[3]:

Gradient:

$$(\nabla p)_{i,,j,k} = \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\delta x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\delta y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\delta z}$$

Divergence:

$$(\nabla \cdot \vec{u})_{i,j,k} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2\delta x} + \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2\delta y} + \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2\delta z}$$

[3]The subscripts i,j,k represent a cell's position in the grid.

Laplacian:

$$\left(\nabla^2 u\right)_{i,j,k} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{\left(\delta x\right)^2} + \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{\left(\delta y\right)^2} + \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{\left(\delta z\right)^2}$$

$$\left(\nabla^2 v\right)_{i,j,k} = \frac{v_{i+1,j,k} - 2v_{i,j,k} + v_{i-1,j,k}}{\left(\delta x\right)^2} + \frac{v_{i,j+1,k} - 2v_{i,j,k} + v_{i,j-1,k}}{\left(\delta y\right)^2} + \frac{v_{i,j,k+1} - 2v_{i,j,k} + v_{i,j,k-1}}{\left(\delta z\right)^2}$$

$$\left(\nabla^2 w\right)_{i,j,k} = \frac{w_{i+1,j,k} - 2w_{i,j,k} + w_{i-1,j,k}}{\left(\delta x\right)^2} + \frac{w_{i,j+1,k} - 2w_{i,j,k} + w_{i,j-1,k}}{\left(\delta y\right)^2} + \frac{w_{i,j,k+1} - 2w_{i,j,k} + w_{i,j,k-1}}{\left(\delta z\right)^2}$$

2.3.2 Numerical simulation

In the field of computational fluid dynamics there are many methods to discretize the Euler equations. In computer animation the method of splitting is the most commonly used (Bridson 2008), (Crane et. al. 2007), (Harris, 2004). In this method each equation is split up into its component parts and each part is solved separately, one at a time (Bridson 2008). Thus, the simulation is divided in steps (or modules) and the output of the first step becomes the input of the second and so forth. It should be noted that the order of the steps is crucial for the correct functioning of the simulation. The basic simulation pipeline is described bellow:

Advection step:

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

External forces step:

$$\frac{\partial \vec{u}}{\partial t} = \vec{f}$$

Pressure step:

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \quad : \quad \nabla \cdot \vec{u} = 0$$

However, it is still not apparent how these modules can be solved. The following subsection will describe a decomposition of the equations that will lead to a form that is compliant to a numerical solution.

2.3.3 Helmholtz-Hodge decomposition theorem

Any vector can be represented as a sum of basis vector components (Harris, 2004). For example, a vector $\vec{u} = (u, v, w)$ can be represented as $\vec{u} = u\hat{i} + v\hat{j} + w\hat{k}$,

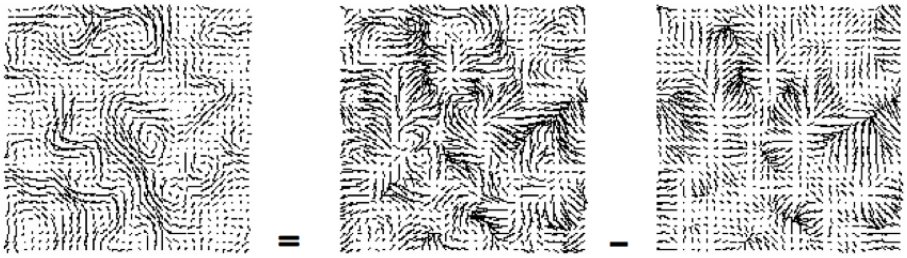

Figure 2.6: The velocity field can become divergent-free when the pressure gradient is subtracted. Figure taken from (Stam, 2003).

where $\hat{i}$, $\hat{j}$ and $\hat{k}$ are unit basis vectors that are aligned on the axis of a Cartesian space. In the same way a vector field can be represented as a sum of vector fields. In a region of space D with a differentiable boundary ∂D, and normal direction n, the Helmholtz-Hodge theorem states that a vector field $\vec{w}$ on D can be uniquely represented as:

$$\vec{w} = \vec{u} + \nabla p \tag{2.10}$$

Where $\vec{u}$ is a divergence-free vector field, parallel to ∂D ($\vec{u} \cdot n = 0$ on ∂D).

Thus, the theorem states that any vector field is the sum of a mass conserving (divergent-free) field and the gradient of a scalar field. This result allows the definition of an operator P which projects the divergent velocity field $\vec{w}$ onto its divergence free part ($\vec{u} = P\vec{w}$) (Stam, 1999). The operator is defined implicitly by applying the divergence operator on both sides of equation 2.10.

$$\nabla \cdot \vec{w} = \nabla^2 p$$

A solution to this equation, which is often called the Poisson-pressure equation, can be used to compute the divergent free velocity field by subtracting the pressure gradient from the divergent velocity field (see figure 2.6):

$$\vec{u} = P\vec{w} = \vec{w} - \nabla p$$

Using the projection operator P, the Euler equations can be rewritten as follows[4]:

$$\frac{\partial \vec{u}}{\partial t} = P\left(-\left(\vec{u} \cdot \nabla\right)\vec{u} + \nu\nabla^2\vec{u} + \vec{f}\right) \tag{2.11}$$

The solution of equation 2.11, over a single time step, can be defined as an operator $\mathbb{S}(\vec{u})$ that takes the current velocity field $\vec{u}$ as its parameter (Harris, 2004). $\mathbb{S}(\vec{u})$ is defined as the composition of three operators: $\mathbb{A}(\vec{u})$ for the

[4] $P\vec{u} = \vec{u}\, and\, P\nabla p = 0$

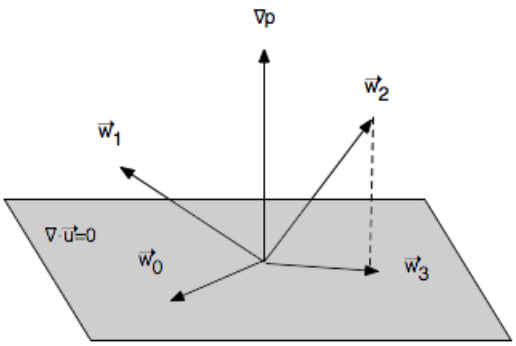

Figure 2.7: Each step of the simulation produces a vector field $\vec{w}$. At the last step, field $\vec{w}_2$ is projected to a divergent-free space using the pressure projection method. Figure modified from (Stam, 1999).

advection step, $\mathbb{F}$ for the external force application step, and $\mathbb{P}$ for the pressure projection step (the operators are applied from right to left):

$$\mathbb{S}(\vec{u}) = \mathbb{P} \circ \mathbb{F} \circ \mathbb{A}(\vec{u}) \tag{2.12}$$

The output of every operator is the new state of the velocity field. When the velocity field reaches step 2, it will most probably be divergent. The pressure projection operator projects the divergent velocity field to its divergent free part (figure 2.7 provides a visualization of this procedure, where every velocity field $\vec{w}$ corresponds to a step of the simulation).

2.3.4 Solving the Poisson pressure equation

The Poisson equation on a 3-dimensional domain Ω with a differentiable boundary $\partial\Omega$ follows the form (Noury et al., 2011):

$$\nabla^2 p = -f \quad on \;\; \Omega \subset \mathbb{R}^3 \tag{2.13}$$

Where p signifies the solution of the Poisson equation and f is known. In most cases, equation 2.13 has no close-form solution for any given right-hand side f and domain Ω. For this reason, a discretization scheme can be used and an approximation of the solution p can be sought in a finite-dimensional space. A very popular discretization method is the finite-volume method, where the average of p is approximated over a set of mesh cells that cover Ω. This mesh is comprised by a set $\mathcal{T}$ of non-overlapping cells C_i:

$$\Omega = \bigcup_{i \in \mathcal{T}} C_i, \quad C_i \cap C_j = \emptyset, \quad \forall i \neq j \in \mathcal{T}$$

Using this discretization in 3 dimensions the Poisson equation in cell $C_{i,j,k}$ can be written as[5]:

$$6p_{i,j,k} - p_{i-1,j,k} - p_{i+1,j,k} - p_{i,j-1,k} - p_{i,j+1,k} - p_{i,j,k-1} - p_{i,j,k+1} = -h^2 f_{i,j,k} \tag{2.14}$$

Or, in matrix form:

$$Ap = f \tag{2.15}$$

Where $A \in \mathbb{R}^{3x3}$ represents a sparse matrix, $p \in \mathbb{R}^3$ represents the vector that contains the cell averages and f represents the right-hand side of equation 2.14.

2.3.4.1 Iterative methods

The sparse matrix A displays two properties that make it invertible with a unique solution: it is symmetric and positive definite:

$$p = A^{-1} f$$

For small grid dimensions, this matrix can be inverted using a direct method (e.g., Gaussian elimination), however, such methods have many drawbacks: they have a $O\left(N^3\right)$ complexity, and require a lot of memory for storing the inverse matrix (which is usually full) (Noury et al., 2011). For this reason iterative methods can be used instead. In such methods an approximation of the Poisson solution can be found using preconditioned iterations:

$$v_{\alpha+1} = \left(I - P^{-1}A\right) v_\alpha + P^{-1} f \tag{2.16}$$

Where, v_α and $v_{\alpha+1}$ are the approximate solutions at iteration α and $\alpha + 1$ respectively, and P represents a preconditioned in the linear system $Ap = f$.

The various available iterative solvers use different preconditioners. However, in most cases, these preconditioners are constructed by decomposing the sparse matrix A into three components: L : the lower-triangular part of the matrix, D : the matrix diagonal, and U : the upper-triangular part of the matrix (see figure 2.8).

2.3.5 Boundary conditions

In the Eulerian discretization scheme that has been used, the fluid is moving inside a 3D cube. However, so far its behavior on the boundaries has not been defined. The simplest boundary condition for the velocity field is the no-stick condition [6] (Bridson, 2008). For a static obstacle the condition specifies that the normal component of velocity must be zero at the boundary:

[5] Where h is the size of the uniform grid cells.

[6] It applies to inviscid fluids.

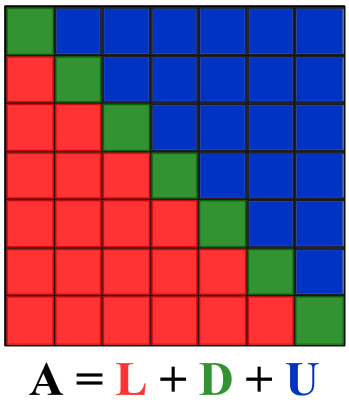

Figure 2.8: Decomposition of the sparse matrix A. L: the low-triangular part, D, the diagonal, and U: the upper-triangular part. Figure modified from (Noury et al., 2011).

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

For, dynamic obstacles, however, the no-stick condition specifies that the normal component of the fluid velocity must be equal to the normal component of the obstacle velocity:

$$\vec{u} \cdot \hat{n} = \vec{u}_{obst} \cdot \hat{n}$$

Using this condition the fluid is free to slip along the tangential direction of the obstacle.

A common boundary condition for the pressure field is the pure Neumann condition where: $\frac{\partial p}{\partial n} = 0$, for $x \in \partial\Omega$ (Harris, 2004). This means, that the rate of change of pressure along its normal component, must equal to zero at the boundary.

2.4 Solving for additional scalar fields

The main application of a fluid solver in computer animation is to use the velocity field to move objects in a realistic fashion (Stam, 1999). In smoke simulation it is required to move smoke particles according to the velocity field. However, it is very computationally expensive to keep track of every particle. For this reason, the smoke particles can be replaced by a continuous density function. This function stores the concentration of smoke particles in every cell of the grid.

Additional scalar fields can also be advected. A general formula that describes the evolution of a quantity in the velocity field is described in equation 2.17:

$$\frac{\partial q}{\partial t} = -\left(\vec{u} \cdot \nabla\right) q + S \tag{2.17}$$

In this equation q defines any quantity that is represented as a scalar field and S defines any sources that increase the amount of this quantity.

Chapter 3

Previous work

This chapter will describe previous works on fluid simulation in the field of computer animation. The examination of these previous implementations has been arranged in chronological order, as each new work has been influenced by previous models.

3.1 Fluid solvers in computer animation

Before describing the research on GPU smoke solvers, it is crucial to examine the structure of some well known fluid solver designed for computer animation applications. For this reason, a short introduction will follow that will describe CPU models that have been implemented.

3.1.1 Foster and Metaxas (1996)

Foster and Metaxas (Foster and Metaxas, 1996) developed one of the first methods in computer animation that numerically solves the incompressible Navier-Stokes equations[1]. They used an explicit finite difference approximation of the equations that could be solved using an explicit Eulerian scheme. However, their method was not stable for any arbitrary time-step and that is why they imposed the following constraint on the velocity field:

$$1 > max[u\frac{\delta t}{\delta x}, v\frac{\delta t}{\delta y}, w\frac{\delta t}{\delta z}]$$

3.1.2 Stam (1999)

Stam (Stam, 1999) was the first to introduce an unconditionally stable method for simulating fluids. He used implicit integration schemes in every simulation

[1]Previous methods used procedural turbulence techniques that produced a pseudo-fluid movement (Stam, 1999).

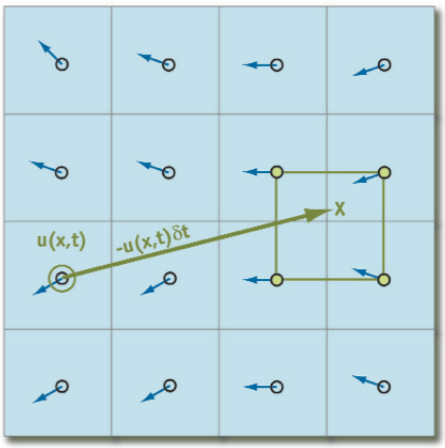

Figure 3.1: The semi-Lagrangian advection method. Figure taken from (Harris, 2004).

step[2] and hence large time-steps could be used, without compromising the stability of the solver. The most important feature of his proposed solver was the advection scheme; the so-called semi-Lagrangian advection, where the method of characteristics is used to solve the partial differential equations. In simple terms, the semi-Lagrangian advection method updates the velocity (or, indeed, any other scalar field) at a grid point $x_{i,j,k}$ by tracing back in time a source value from the grid. To intuitively understand this, the advection can be thought of as a Lagrangian process where particles move through the velocity field. At any time t a particle can be moved from position $x_{i,j,k}$, using the negative velocity $\vec{u}_{i,j,k}(t)$ (the velocity at grid position $x_{i,j,k}$). Using the negative velocity, will move the particle in its position one time-step ago. However, the particle will most likely end up in a position in-between grid cells or even outside the grid. In order to find an approximate value from that position, trilinear interpolation can be used on the closest neighboring cells. The resulting value will then be used as the updated value in position $x_{i,j,k}$. Figure 3.1 describes this process.

Although this advection scheme is unconditionally stable, it introduces numerical dissipation to the simulation (Bridson, 2008). In order to overcome this problem, higher accuracy interpolation schemes can be used.

3.1.3 Fedkiw et al. (2001)

Fedkiw et al. (Fedkiw et al., 2001) described a method for the visual simulation of smoke. Their solver has many similarities with Stam's method; however, in order to simulate the movement of the smoke particles, they have added a

[2]With the only exception of the force application step, where explicit integration can be safely used.

temperature and a density field. Both these fields affect the evolution of the velocity field: dense smoke has a tendency to fall downwards, because of gravity, while hot smoke tends to rise, due to buoyant forces. In order to accommodate these interactions, they used a simple model, where a buoyant force proportional to the smoke velocity and temperature is added to the velocity field:

$$f_{buoy} = -\alpha\rho z + \beta(T - T_{amb})z \tag{3.1}$$

Where ρ represents density, T represents temperature, T_{amb} represents the ambient temperature of the air, z is the buoyancy direction and α and β are two positive constants.

3.2 GPU implementations

In this section three GPU based smoke solvers will be described.

3.2.1 Harris (2004)

Harris (Harris, 2004) described one of the first GPU methods for stable fluid simulation on the GPU. His 2D solver follows the method proposed by Stam. For the representation of the simulation fields 2D textures are used and the simulation kernels run on fragment programs. Every simulation step is preformed by rendering a 2D quad and executing the appropriate fragment program. The fragment programs that are used by the solver are described below:

- **Advect:** this program performs the semi-Lagrangian advection step.
- **Diffuse:** this program is responsible for the viscous diffusion step. However, as stated before, this step can be skipped in smoke simulation.
- **Apply force:** in this module, force is applied to the velocity field.
- **Divergence:** this program calculates the divergence of the velocity field. The divergence field is used in the Poisson solver, in order to perform the pressure projection step.
- **Jacobi:** this program uses the Jacobi iteration method to solve the Poisson equations.
- **Subtract gradient:** this kernel calculates the pressure gradient using finite differences, and it subtracts it from the velocity field. This is the part of the simulation where the incompressibility condition is enforced.
- **Boundary:** the simulation boundary values are stored in 4 line primitives. These primitives hold values for both the velocity and the pressure boundary conditions.

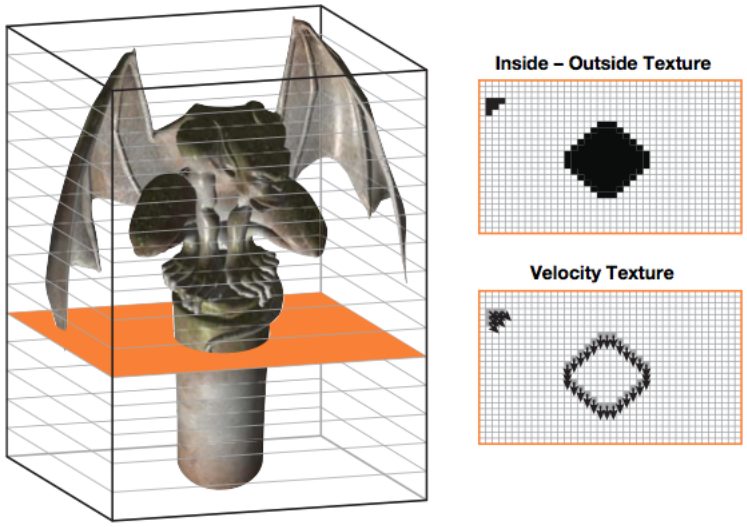

Figure 3.2: Voxelization of complex geometry. The collision and velocity texture maps are displayed on the right. Figure taken from (Crane et al., 2007).

3.2.2 Crane et al. (2007)

Crane et al. (Crane et al., 2007) expanded Harris' method and introduced a more advanced solver that can be used in any rasterization-based rendering paradigm. One of the new features that they introduced to their model was the handling of dynamic obstacles. They use a voxelization scheme, where every geometry can be mapped to a voxel map. They store two separate texture maps for this operations: one for the collision data (where a voxel is marked either as occupied or as free) and one for the velocity data (where the obstacle velocity data are stored on the object's boundary) (see figure 3.2). This method facilitates the enforcement of the no-stick boundary conditions and thus allows the fluid to flow around a moving obstacle.

3.2.3 Rideout (2011)

Rideout (Rideout, 2011) created a solver that is based on the two previous implementations. It uses frame-buffer operations for the execution of the simulation programs, and every field is represented as a 3D texture. The solver also features an impulse application step where smoke density and temperature are introduced to the simulation. The managing of both boundary and obstacle conditions is handled using voxel maps (similar to the approach of Crane et al.) so all the simulations collision and velocity data are stored in one texture. The pipeline of this application is displayed in figure 3.3:

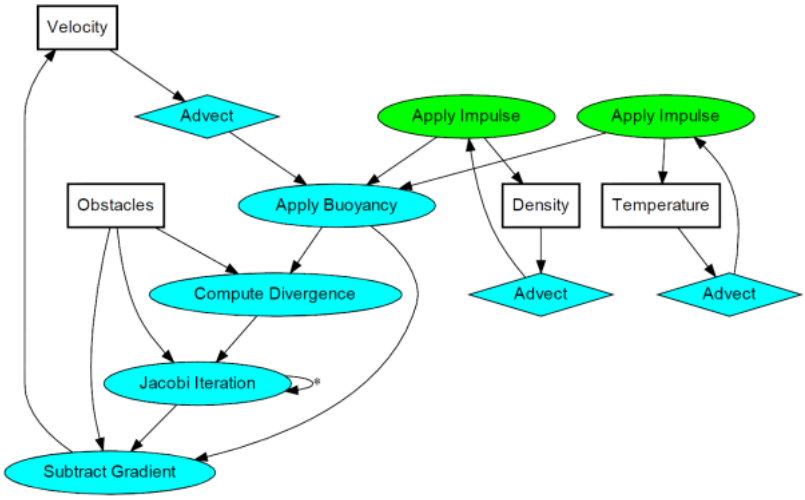

Figure 3.3: The application diagram of Rideout's implementation. Figure taken from (Rideout, 2010).

Chapter 4

Implementation

In this chapter the implemented smoke simulation application is described. Firstly, the OpenCL standard will be discussed and subsequently the design and implementation of the application will be analyzed.

4.1 OpenCL overview

OpenCL is an open standard for programming in heterogeneous systems, that is, systems comprised of multiple parallel processing units (referred to as devices in the OpenCL lingo) like multicore CPUs and GPUs (Noury et al., 2011), (Munshi et al., 2011). The OpenCL pipeline can be divided into two distinct parts. In the first part, a CPU based host runs OpenCL API commands that create memory objects on the devices, manage the execution of parallel programs and control the interactions between host and devices (in the same way that the OpenGL/GLSL pipeline works) (Noury et al., 2011). In the second part, parallel programs (called compute kernels) written in the OpenCL C language are executed on the devices (in the same way as shader programs).

4.1.1 Execution model

When the host issues a kernel execution command, the OpenCL runtime system generates an integer index space (Munshi et al., 2011). This space can have an N-dimension range of values and thus it is referred to as the kernel NDRange. The compute kernel is executed once for each point in the NDRange. Each thread of the executing kernel is called a work-item and is uniquely identified by its index space coordinates. These coordinates constitute the global ID of the work-item.

The NDRange is divided into groups of work-times; these thread ranges are called work-groups. Every work-group has an ID in the NDRange and every work-item in the group has a local ID defining its local coordinates. This division scheme is described in figure 4.1.

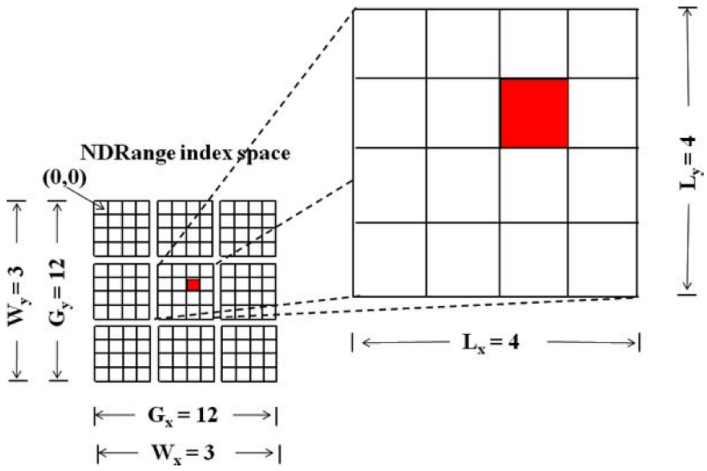

Figure 4.1: A 2D NDrange displaying all the possible work-item coordinate categorizations, G: global coordinates, W: work-group coordinates, L: local coordinates. Figure taken from (Munshi et al., 2011).

4.1.2 Memory model

OpenCL specifies two kinds of memory objects: buffer objects and image objects. Buffers are contiguous blocks of memory indexed by 1D coordinates and composed of any available OpenCL data type (int, float, half, double, int2, int3, int4, float2, ...) (Noury et al., 2011). Any data structure can be mapped onto a buffer and its data can be accessed through pointers (Munshi et al., 2011).

Images have many similarities with OpenGL textures: they both support an automatic caching mechanism and their data are accessed through sampler objects (Noury et al., 2011). Samplers can filter the image data using multilinear interpolation, and, furthermore, they allow the configuration of out of bound access behavior.

In addition to the memory object types, the OpenCL memory model specifies 5 regions, where data can be allocated (Munshi et al., 2011):

- **Host memory:** this region is exclusively visible by the host. It is used for interactions between the host and OpenCL objects.
- **Global memory:** this region gives access to reading and writing operations to all the work-items in every work-group. Global memory connects the host and the employed devices by enqueuing reading and writing operations. It has a large scale capacity, but it also has high latency.
- **Constant memory:** it represents a region of global memory that stays constant through a kernel execution (similar to a shader uniform). Data stored in constant memory can only be read by the compute kernels.

- **Local memory:** this region can be accessed by a work-group. It has a very limited size, but it provides faster data access than global memory.
- **Private memory:** this region is visible only to a single work-item.

4.1.3 OpenCL / OpenGL interoperation

OpenCL supports an interoperation model with OpenGL (Noury et al., 2011). Using the appropriate extension, the interoperation model allows the creation of OpenCL buffers and images using existing OpenGL buffers and textures. However, when OpenCL commands are to be issued to these reference objects, pending OpenGL operations must be completed (or if the appropriate extension is supported, synchronization methods can be employed).

4.1.4 Compute engine

In the implemented application, a dedicated class has been created for the managing of the OpenCL context and all OpenCL commands, the compute engine class [1]. The compute engine holds a map structure for every program object, kernel, memory object and image sampler that is created in the OpenCL context. It also creates a command queue for every available compute device and handles all the host API calls.

4.2 Memory objects

The simulation output by itself has no visual importance until it is rendered. As it will be later detailed, the application employs a gas solver that is implemented in OpenCL. However, in order to allow the interoperation with OpenGL, the appropriate memory objects must be created. The following subsections will detail the memory objects that are used by the application.

4.2.1 Textures and buffers

In order to facilitate the use of OpenGL memory objects, dedicated wrapper classes have been created. The buffer object class allows the creation and handling of buffer objects, such as array buffers and pixel buffers. The class manages the binding of the buffers to selected OpenGL targets and handles data loading operations.

For the creation of texture objects, 4 classes have been created. A texture base class with the subclasses: texture 1D, texture 2D and texture 3D. In the same fashion with the buffer object class, the texture classes manage all the texture object functionalities (see figure 4.2).

[1] This class is modified from (Apple Inc., 2011).

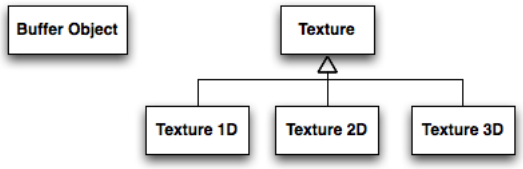

Figure 4.2: The application memory objects.

4.2.1.1 Texture image unit stack

When a texture object is created, it must be attached to a texture image unit. The number of the available image units depends on the GPU capabilities and the OpenGL implementation. Although it is possible to attach two or more textures on the same unit, when these textures are used in the same shader only one of them will be sampled (and it is undefined which one). In order to facilitate the creation of textures in a rendering context,[2] a utility class has been created: the texture image unit stack. It is a stack structure that stores all the available texture image units and returns the next available unit.

4.2.2 Ping-pong volumes

Texture objects can be ideal candidates for the representation of the simulation fields. They internally support trilinear interpolation and allow configurable out of bounds behavior, features which are fundamental in the gas solver functionality. However, currently most OpenCL implementations do not support in-kernel writing operation on 3D images. For this reason, a turnaround can be employed in order to overcome this restriction. A referenced pixel buffer object (PBO) can be used as the writing target in the compute kernels: at the end of each solver cycle, the PBO data can be uploaded to a 3D texture by enqueuing a copy command. Although the copy command adds overhead to the application, it still remains a GPU located operation, so the simulation data never actually leave the GPU. In order to accommodate for these reading and writing operations, a ping-pong volume class has been created. It is a volume object that uses a PBO for all writing operations and a 3D texture for all reading operations. It uses a swap method for the uploading of the PBO data to the texture. In this way, the benefits of 3D texture data access in OpenCL are preserved by adding an extra step to the data writing operation. In order to reference the OpenGL memory objects of the volume, two references must be created in OpenCL: a 3D image reference and a buffer reference.

[2] A context where multiple shaders share a common set of resources.

Algorithm 4.1 The advection kernel.

```
// check for obstacles and boundary
IF (cell[i,j,k] == SOLID) THEN
{
  field[i,j,k] = 0;
  EXIT
}

// find cell coordinates "back in time"
[i0,j0,k0] = [i,j,k] - dt * V[i,j,k];

field[i,j,k] = dissipationFactor * V[i0,j0,k0];
```

Algorithm 4.2 The buoyancy application kernel.

```
IF (T[i,j,k] > ambientTemperature) THEN
{
  V[i,j,k] += (
                dt * (T[i,j,k] - ambientTemperature)
                * buoyancyLift - D[i,j,k] * gasWeight
              ) * buoyancyDirection;
}
```

4.3 The gas solver

This section will detail all the different modules of the implemented gas solver.

4.3.1 Advection

The advection kernel performs semi-Lagrangian advection on the velocity, temperature and density fields. In order to find the coordinates of the particle one time-step back, forward Euler integration is used. Algorithm listing 4.1 displays the pseudocode of the kernel where cell[i,j,k] corresponds to a grid position at point (i,j,k), field[i,j,k] corresponds to any field value at the current position and V stands for the current velocity.

4.3.2 Buoyancy application

The buoyancy application kernel is responsible for the buoyant force that makes dense smoke sink and hot smoke rise. The buoyant force is applied to cells where the temperature is greater than the ambient temperature. Algorithm listing 4.2 displays the pseudocode of the kernel, where T represents the temperature field and D represents the density field.

Algorithm 4.3 The impulse application kernel.

```
IF (cel[i,j,k] == INSIDE-SPLAT-REGION) THEN
{
  D[i,j,k] = densityAmount;
  T[i,j,k] = temperatureAmount;
}
```

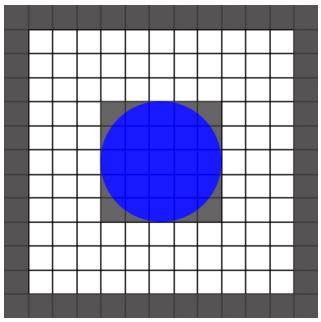

Figure 4.3: Iso-surface voxelization for a sphere surface. The obstacle object contains an instance of the simulation boundary.

4.3.3 Impulse application

The impulse application kernel acts as a source for temperature and smoke density. The sources are added using a Gaussian splat; that is, cells that are inside the splat radius will be assigned a specified amount of temperature and density. Algorithm listing 4.3 displays the kernel pseudocode:

4.3.4 Adding obstacles

Obstacles are added into the simulation, using implicit surfaces. In order to voxelize the surfaces, an implicit surface is sampled on the simulation grid; if a cell contains an iso-value that is less than zero, then this cell is marked as solid. In order to merge the boundary checks with the obstacle collision detection, every obstacle object contains an instance of the simulation boundary (see figure 4.3). The gas solver also supports the use of dynamic obstacles; however, moving obstacles have not yet been implemented in the application and all obstacles are assumed to be stopped with zero velocity (the boundary is also considered an obstacle with zero velocity).

4.3.5 Pressure projection

In the pressure projection step, the differentiation operators of the incompressible Euler equations are approximated using central differences. In order to

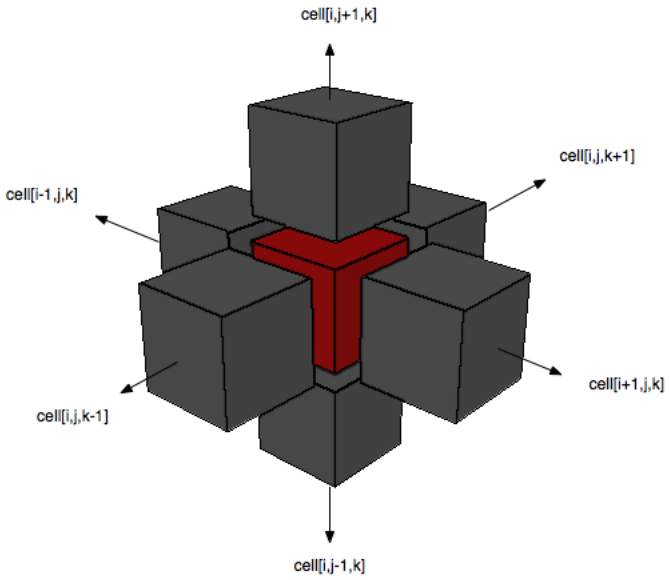

Figure 4.4: A 3D stencil, used for the calculation of central differences. The red cell represents the current cell. Figure modified from (Noury et al, 2011).

calculate these finite differences, a 3D stencil can be used for the allocation of the neighboring cells (see figure 4.4).

The pressure projection step involves 3 kernels:

- **Divergence computation kernel:** where the divergence of the velocity field is calculated and the pressure field is set to zero.
- **Poisson solver:** where the Poisson-pressure equation is solved, using the divergence of the velocity field.
- **Pressure gradient subtraction kernel:** where the pressure gradient is calculated and subtracted from the divergent velocity field.

4.3.5.1 Divergence computation

In the divergence computation kernel, the divergence of the velocity field is calculated, using central differences. Moreover, an obstacle check is being performed and if a neighboring cell is occupied by a solid, the obstacle velocity is used on that component. At the end of the program, the pressure field is set to zero, in order to be ready for processing by the Poisson solver. Algorithm listing 4.4 displays the kernel pseudocode, where O represents the obstacle array and P represents the pressure field.

Algorithm 4.4 The divergence computation kernel.

```
// find velocity stencil (up, down, right, left, front and back)
Vu = V[i,j+1,k];
Vd = V[i,j-1,k];
Vr = V[i+1,j,k];
Vl = V[i-1,j,k];
Vf = V[i,j,k+1];
Vb = V[i,j,k-1];

// find obstacle stencil
Ou = O[i,j+1,k];
Od = O[i,j-1,k];
Or = O[i+1,j,k];
Ol = O[i-1,j,k];
Of = O[i,j,k+1];
Ob = O[i,j,k-1];

// if cell is solid use obstacle velocity
IF (Ou == SOLID) THEN { vU = oU; }
IF (Od == SOLID) THEN { vD = oD; }
IF (Or == SOLID) THEN { vR = oR; }
IF (Ol == SOLID) THEN { vL = oL; }
IF (Of == SOLID) THEN { vF = oF; }
IF (Ob == SOLID) THEN { vB = oB; }

// compute divergence field
divergence[i,j,k] = (Vr.x - Vl.x + Vu.y - Vd.y + Vf.z - Vb.z) / 2*cellSize;
P[i,j,k] = 0;
```

4.3.5.2 Pressure gradient subtraction

The pressure gradient subtraction kernel calculates the pressure gradient and subtracts it from the velocity field. It also sets the boundary conditions by using the neighboring pressure on solid cells, and by enforcing the no-stick condition. Algorithm listing 4.5 displays the pseudocode of the kernel.

4.3.5.3 Poisson solver

A common iterative method for the solution of the Poisson-pressure equation is the Jacobi iteration method (Noury et al., 2011). This method can be parallelized easily on a GPU implementation and has been used by Harris, Crane et al. and Rideout (Rideout, 2011), (Crane et al., 2007), (Harris, 2004). The Jacobi method uses the diagonal of the sparse matrix A (see equation 2.15) as its preconditioner $P_j = D$. Thus, equation 2.16 can be written as [3]:

$$v_{\alpha+1} = \left(I - \frac{1}{6}A\right) v_\alpha + \frac{1}{6}f \tag{4.1}$$

Equation 4.1 can be easily translated into a compute kernel (see algorithm listing 4.6).

4.3.6 Adding turbulence

The solvers allows additional control over the simulation by adding procedural noise to the velocity field. The periodic noise function adds randomness to the buoyancy direction and allows the configuration of a trigonometric driving function (sine, cosine, and tangent functions). This method provides additional control over the fluid motion and enhances the smoke detail.

4.4 Real-time rendering

The most interesting output in the smoke simulation is its density field. In this section, two volume rendering technique that produce a visual output of the simulation data will be described.

4.4.1 The marching cubes algorithm

The marching cubes algorithm was originally developed for the visualization of medical data (mainly data from MRI or CT scans) (Lorensen and Cline, 1987). However, it can also be used for the visualization of any scalar field. The algorithm performs a polygonization operation on constant density data, which are represented in the form of a 3D array. The algorithm progresses in the following steps:

[3] In a 3D Laplace matrix the diagonal is equal to 6, thus, $D^{-1} = \frac{1}{6}I$ (Noury et al., 2011).

Algorithm 4.5 The pressure gradient subtraction kernel.

```
IF (cell[i,j,k] == SOLID) THEN
{
  V[i,j,k] = OBSTACLE VELOCITY;
  EXIT
}

// find pressure stencil
Pu = P[i,j+1,k];
Pd = P[i,j-1,k];
Pr = P[i+1,j,k];
Pl = P[i-1,j,k];
Pf = P[i,j,k+1];
Pb = P[i,j,k-1];

// find obstacle stencil
Ou = O[i,j+1,k];
Od = O[i,j-1,k];
Or = O[i+1,j,k];
Ol = O[i-1,j,k];
Of = O[i,j,k+1];
Ob = O[i,j,k-1];

// use center pressure for solid cells
Vobstacle = (0,0,0);
Vmask     = (1,1,1);

IF (Ou == SOLID) THEN { Pu = P[i,j,k]; Vobstacle.y = Ou.z; Vmask.y = 0; }
IF (Od == SOLID) THEN { Pd = P[i,j,k]; Vobstacle.y = Od.z; Vmask.y = 0; }
IF (Or == SOLID) THEN { Pr = P[i,j,k]; Vobstacle.x = Or.y; Vmask.x = 0; }
IF (Ol == SOLID) THEN { Pl = P[i,j,k]; Vobstacle.x = Ol.y; Vmask.x = 0; }
IF (Of == SOLID) THEN { Pf = P[i,j,k]; Vobstacle.z = Of.x; Vmask.z = 0; }
IF (Ob == SOLID) THEN { Pb = P[i,j,k]; Vobstacle.z = Ob.x; Vmask.z = 0; }

// enforce the no-stick boundary condition
Vold = V[i,j,k];
gradient = (Pr - Pl, Pu - Pd, Pf - Pb) * gradientScale;
Vnew = Vold - gradient;
V[i,j,k] = (Vmask * Vnew) + Vobstacle;
```

Algorithm 4.6 The Poisson solver kernel.

```
// find pressure stencil
Pu = P[i,j+1,k];
Pd = P[i,j-1,k];
Pr = P[i+1,j,k];
Pl = P[i-1,j,k];
Pf = P[i,j,k+1];
Pb = P[i,j,k-1];

P[i,j,k] = (Pl + Pr + Pd + Pu + Pf + Pb - (h*h) * f[i,j,k]) / 6;
```

1. The scalar field is being examined in respect to a user-defined threshold value. When implicit surface modeling is considered, this value should be 0, so that it defines the points that lie on the surface.

2. The finite volume of the discretization is comprised of cubes that form a grid structure. For every cube, a set of vertices that are intersected by the surface are calculated. These vertices are calculated on the edges of each cube. The exact position of a vertex is configured through a linear interpolation scheme which follows the sampled values. This calculation is performed with the use of an edge look-up table.

3. For the actual triangulation of the vertices, a triangulation look-up table is used (Bourke, 1994). This table defines the sequence of the vertices in a manner that can be rendered by a graphics API (usually counter clockwise).

4. Finally, the normals for each vertex are computed by calculating the surface gradient at each cube corner and by linearly interpolating between these gradient values.

Due to the fact that a cube has 8 vertices, there are totally $2^8 = 256$ ways by which a surface can intersect a cube. However, because of two symmetrical properties of the topology of the cube these cases can be reduced to 14 unique patterns. These patterns are depicted in figure 4.5:

The class of each cube can be represented using a single byte. The byte is initialized with 0 and for each vertex that satisfies the threshold condition (an iso value in the case of implicit surfaces) we assign a 1. The numbering of the vertices and the edges of each cube are depicted in figure 4.6.

The edges that are intersected by the surface can be determined by calculating the class (index) of the cube. In order to find the exact position of every triangle vertex, a linear interpolation can be performed with two weights: each weight corresponds to the sampled value at the two vertices that belong to an intersected edge. The intersected edges are determined by an edge, look-up table. The edge table returns a 12 bit number for every cube class (256 cells),

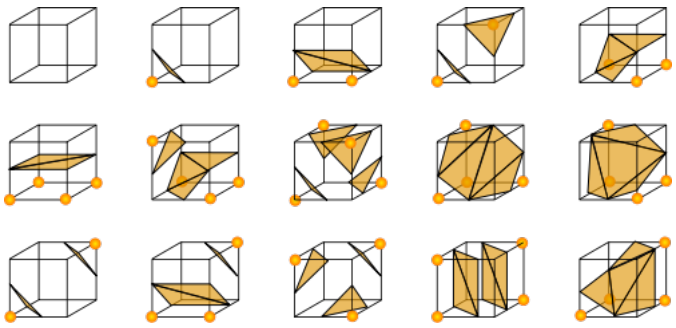

Figure 4.5: The marching cubes cases. Figure taken from (Favreau, 2006).

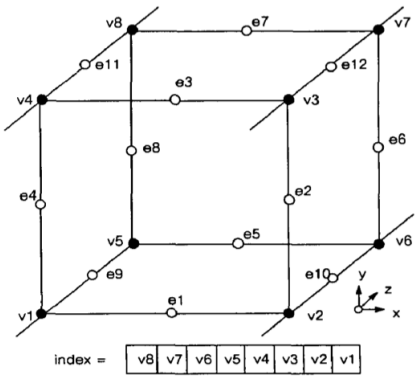

Figure 4.6: Cube edge and vertex indices. Figure taken from (Lorensen and Cline, 1987).

each bit corresponds to an edge; its value is zero, if the edge is not intersected by the iso-surface, and one, if the edge is intersected by the iso-surface (Bourke, 1994).

The triangulation table holds the sequence of vertex indices. It is a 2D table which has 256 rows (one for each cube class) and 16 rows, since the maximum number of produced triangles is 5. The last row marks the end of a triangle sequence (for the classes that produce 5 triangles).

4.4.1.1 Surface shading

The last part of the marching cubes algorithm calculates a normal for each vertex. The normal vector is required if any BRDF (Bidirectional Reflectance Distribution Function) model is to be used. One of the most common methods to compute the iso-surface normals is to use the gradient of the field function, which is actually the normal of the surface (Bloomenthal, 2001), (Nielson et al., 2002) (see equation 4.2).

$$N(x,y,z) = \nabla F(x,y,z) = \left(\frac{\partial F}{\partial x}(x,y,z), \frac{\partial F}{\partial y}(x,y,z), \frac{\partial F}{\partial z}(x,y,z) \right) \quad (4.2)$$

Consequently, a numerical differentiation method can be employed for the approximation of the spatial derivatives. This calculation must be performed for every cube's vertex and the normal for every triangle vertex can be found through interpolation along the cube edges.

4.4.1.2 GPU marching cubes

Programmable geometry shaders are a relatively new addition to the graphics pipeline. They take place after vertex processing and before viewport clipping (Wright et al., 2010). Unlike vertex and fragment (or pixel) shaders, where each processing cycle accesses only one unit (vertex or fragment), geometry shaders have access to whole primitives (e.g, lines, triangles). Geometry shaders can perform a certain amount of tessellation (their capabilities are not immense, however) by producing new geometry and they can also discard geometry.

One of the first implementations of the marching cubes algorithm was presented in SIGGRAPH 2006 by Tariq (Tariq, 2006) (for DirectX10). A similar approach has been also demonstrated by Crassin (Crassin, 2007) (for openGL).

The VolumeMesher class encapsulates all the functionality that is required for the rendering of a scalar field, using the GPU marching cubes algorithm [4]. The look-up tables and the density field are stored in textures and only the grid cell centers are uploaded to the vertex processor. Apart from the configuration

[4] It also supports the dividing cubes algorithm, which renders the whole voxel cubes that are occupied by the field.

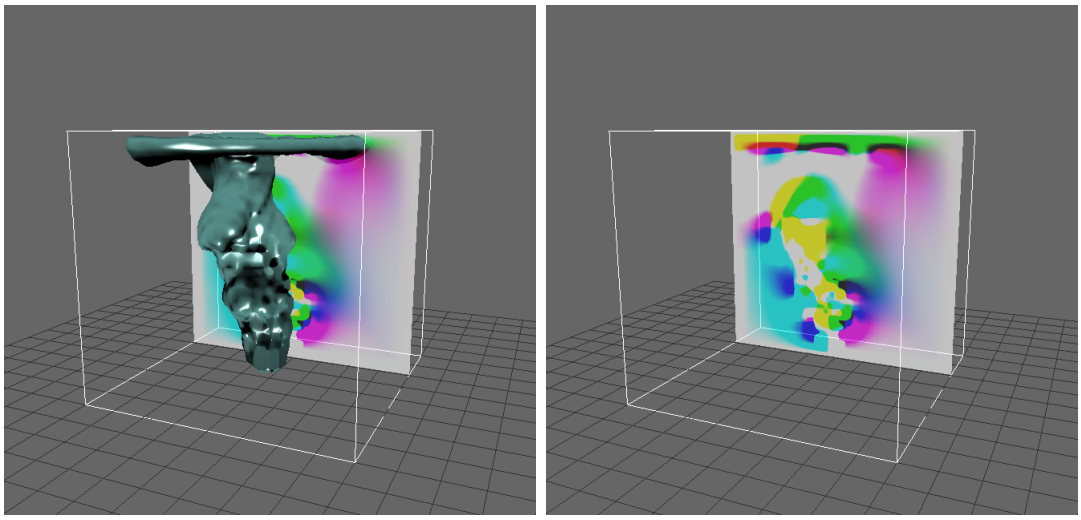

Figure 4.7: A slice of the velocity field rendered a 2D plane. The vector values are biased and clamped to the range [0,1].

of the iso-level threshold, the class allows the selection of sampling channels from the textures, so that vector field components can also be rendered.

4.4.2 Volume slice rendering

At the initial development steps of the application, it was important to visualize more that one field at a time, because the printing of a field's data does not provide intuitive and fast feedback. For this reason, the volume slicer class was created. The volume slicer renders one slice of a 3D texture in a 2D plane. The user can move the plane's position and the program samples the texture from the relevant position (see figure 4.7).

4.5 Application structure

The application's main functionality is en-captured in the fluid engine class. The OpenGL window has an instance of the fluid engine and sends command request according to the user interface input. Figure 4.8 shows the main structure of the application.

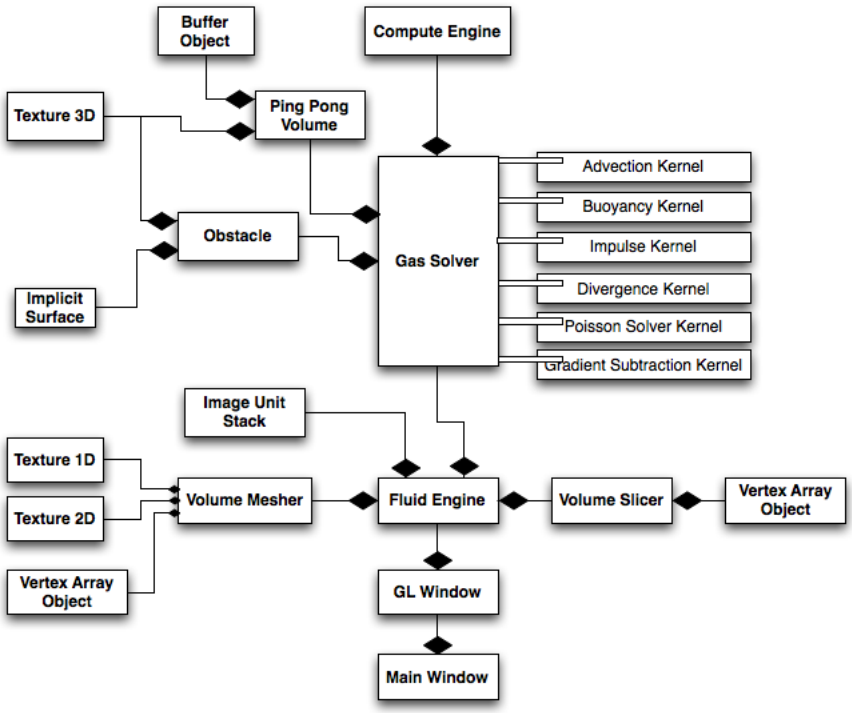

Figure 4.8: The application class diagram.

Chapter 5

Conclusion

In this last chapter the outcomes of the proposed application are discussed, pinpointing issues that were encountered.

5.1 Results

The solver can produce interactive and realistic smoke flow at a resolution of 32x32x32. Figure 5.1 demonstrates an example simulation at this resolution:

Implicit surface obstacles can be used in order to sculp the smoke according to their shape. Figures 5.2 and 5.3 show interactions of smoke with static obstacles.

Interesting results can be attained when noise is introduced to the velocity field. Figure 5.4 exhibits a turbulent buoyant force that is driven by a trigonometric function (pure noise, sine, cosine and tangent functions respectively).

A grid resolution of 64x64x64 produces especially detailed results. Figure 5.5 shows two example smoke simulations at that resolution.

5.2 Efficiency

The application was developed on a Macbook pro using an NVIDIA GeForce 320M, with 256 MB of virtual memory. Although the speed of the application was not extraordinary high, it could still provide real time interaction with the system. Table 5.1 demonstrates the average frames per second for 4 different grid resolutions.

5.3 Known issues

One problem that has not been resolved in the final implementation is related with the advection step. The advected fluids display a slight tendency to follow the vector (-1,-1,-1). A solution to this problem could be a more accurate

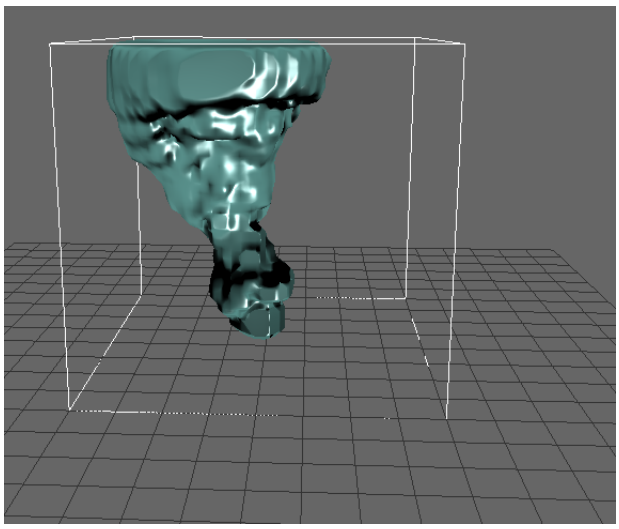

Figure 5.1: A 32x32x32 grid example simulation.

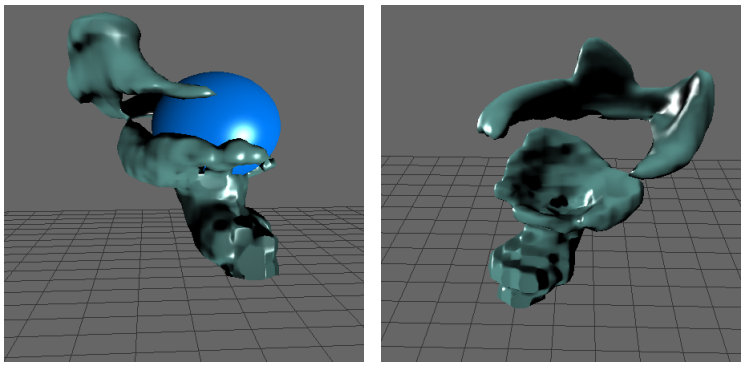

Figure 5.2: Smoke interacting with a sphere.

Grid resolution	FPS
8x8x8	38
16x16x16	20
32x32x32	7
64x64x64	3

Table 5.1: The application speed on different grid resolutions.

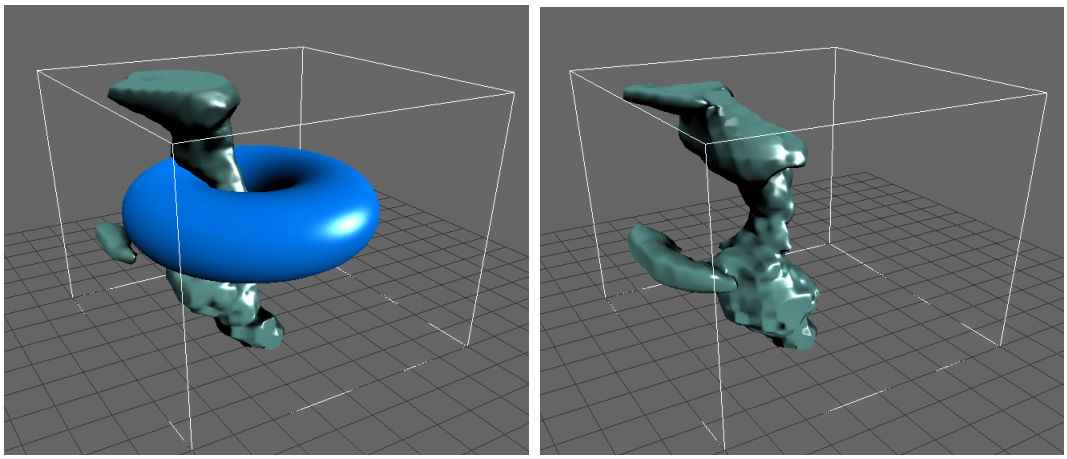

Figure 5.3: Smoke interacting with a torus.

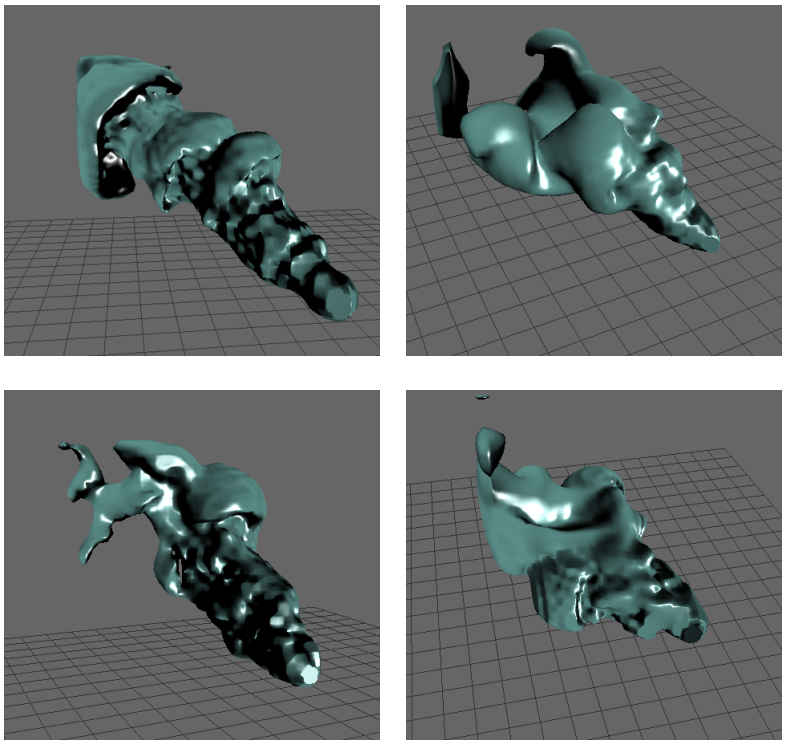

Figure 5.4: Interesting fluid motion can be achieved using the periodic noise function. The figures show an example smoke simulation using the standard noise function, the sine driven function, the cosine driven function and the tangent driven function.

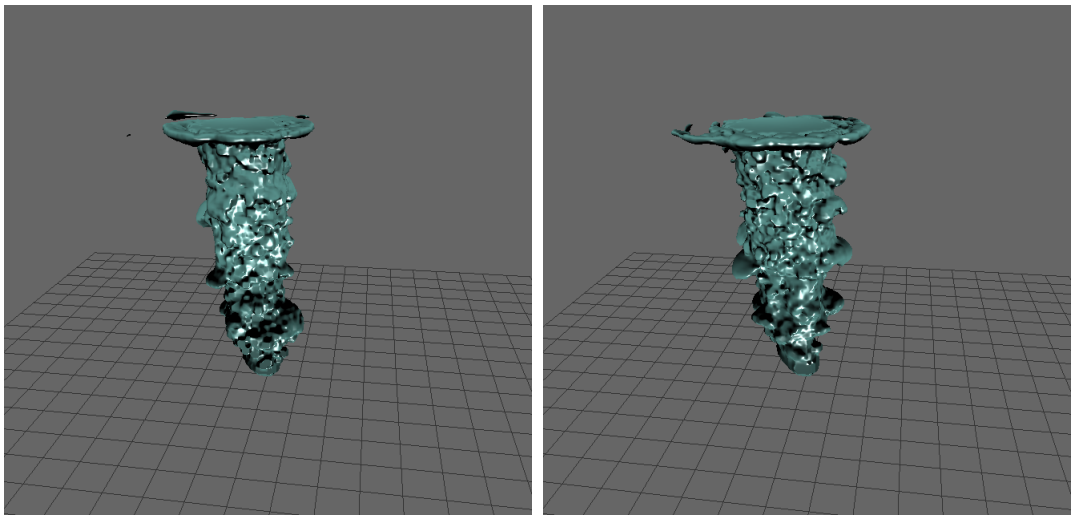

Figure 5.5: Two example simulations at a resolution of 64x64x64. The right picture was produced with the use of the periodic noise method.

advection scheme, such as the MacCormack method. However, this deviation is not always visible and it dissolves if the external forces are strong enough.

Another issue that is apparent in the application is the polygonal representation of the fluid. Even though the mesh is accurate and it is rendered fast, the outcome does not look like a gaseous fluid; it rather appears more like a plastic solid (like clay). However, the focus of this thesis has been placed more on the simulation process than on photorealistic rendering.

5.4 Future work

Future work might concentrate on the implementation of a GPU ray-caster, so that the generated smoke may look more like a gaseous fluid. Additionally, it would be interesting, if a faster Poisson solver could be implemented (for example, a multi-grid solver).

Chapter 6

Bibliography

1. Apple Inc., 2006. OpenCL procedural grass and terrain example. Cupertino: Apple Inc.. Available from: http://developer.apple.com [Accessed 25 May 2011].

2. Biały, S., 2010. Scalar field [figure]. http://en.wikipedia.org. Available from: http://upload.wikimedia.org/wikipedia/en/f/fe/Scalarfield.jpg [Accessed 30 May 2011].

3. Bloomenthal, J., 2001. Implicit surfaces. In: Henderson, H., ed., Encyclopedia of Computer Science and Technology. New York, NY, USA: Marcel Dekker, Inc..

4. Bourke, P., 1994. Polygonising a scalar field. Cupertino: http://paulbourke.net Available from: http://paulbourke.net/geometry/polygonise [Accessed 1 April 2011].

5. Bridson, R., 2008. Fluid simulation for computer graphics. Natick, MA, USA: A K Peters.

6. Bridson, R. and Muller-Fischer, M., 2007. Fluid simulation: SIGGRAPH 2007 course notes. ACM SIGGRAPH.

7. Crane, K., Llamas, I., and Tariq, S., 2007. Real-time simulation and rendering of 3D fluids. In Nguyen, H., editor, GPU Gems 3. Indiana, IN, USA: Addison-Wesley Professional.

8. Crassin, C., 2007. OpenGL geometry shader marching cubes. http://paulbourke.net Available from: http://paulbourke.net/geometry/polygonise [Accessed 1 April 2011].

9. Favreau, J., M., 2006. Marching cubes cases [figure]. http://commons.wikimedia.org. Available from: http://en.wikipedia.org/wiki/File:MarchingCubes.svg [Accessed 3 April 2011].

10. Fedkiw, R., Jos, S., and Henrik, J., 2001. Visual simulation of smoke. In SIGGRAPH '01: Proceedings of the 28th annual conference on computer graphics and interactive techniques, pages 15–22. ACM.

11. Foster, N., and Metaxas, D., 1996. Realistic animation of liquids. Graphical models and image processing, 58(5).

12. Griebel, M., Dornsheifer, T., and Neunhoeffer, T., 1997. Numerical Simulation in Fluid Dynamics: A Practical Introduction. (Monographs on Mathematical Modeling and Computation). SIAM: Society for Industrial and Applied Mathematics.

13. Harris, M. J., 2004. Fast fluid dynamics simulation on the GPU. In R. Fernando, ed., GPU Gems: Programming Techniques, Tips and Tricks for Real- Time Graphics, pages 637–665. Indiana, IN, USA: Addison-Wesley Professional.

14. Lorensen W. E., and Cline, H. E., 1987. Marching cubes: A high resolution 3D surface construction algorithm. ACM Siggraph Computer Graphics, 21(4) : 163– 169.

15. Munshi, A., Gaster, B., Mattson, T., Fung, J., and Ginsburg, D., 2011. OpenCL programming guide. Addison-Wesley Professional.

16. Nielson, G. M., Huang, A., and Sylvester, S., 2002. Approximating normals for marching cubes applied to locally supported isosurfaces. VIS 2002. IEEE, pages 459–466.

17. Noury, S., Boivin, S., and Le Maitre, O, 2011. A fast Poisson solver for OpenCL using Multigrid methods. In Engel, W., ed., GPU Pro 2. A K Peters.

18. Rideout, P., 2010. Simple fluid simulation. http://prideout.net/blog/. Available from: http://prideout.net/blog/?p=58 [Accessed 28 May 2011].

19. Rideout, P., 2011. 3D Eulerian grid. http://prideout.net/blog/. Available from: http://prideout.net/blog/?p=66 [Accessed 28 May 2011].

20. Stam, J., 1999. Stable fluids. In Proceedings of the 26th annual conference on Computer graphics and interactive techniques, SIGGRAPH '99, pages 121– 128, New York, NY, USA: ACM, Press/Addison–Wesley Publishing Co.

21. Stam, J., 2003. Real-time fluid dynamics for games. In Proceedings of the Game Developer Conference, pages 1–17.

22. Tariq, T., 2006. DirectX10 Effects. In SIGGRAPH 2006, page 626. Wordware Publishing.

23. Van Dyke, M., 1988. Album of Fluid Motion. Parabolic Press, Inc., 4th edition.

24. Weisstein, E. W., 2011. Vector field plot. MathWorld - A Wolfram web resource. Available from: http://140.177.205.23/VectorField.html [Accessed 1 June 2011].

25. Wicke, M., Keiser, R., and Gross, M., 2007. Fluid simulation. In Gross, M. and Pfister, H., eds., Point-based graphics. Morgan Kaufmann.

26. Wright, R. S. , Haemel, N. , Sellers, G., and Lipchak, B., 2010. OpenGL SuperBible: Comprehensive Tutorial and Reference. Addison-Wesley Professional, 5th edition.