NTNU
Det skapende universitet

Animering av eksplosjoner i sanntid

Lars Andreas Ek
Rune Vistnes

Master i datateknikk
Oppgaven levert: Juli 2007
Hovedveileder: Torbjørn Hallgren, IDI
Biveileder(e): Odd Erik Gundersen, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Oppgaven går ut på å oppnå visuelt tilfredsstillende tredimensjonale eksplosjoner i sanntid (mer 30 bilder i sekundet) ved fysisk basert simulering og visualisering. Eksplosjonen skal også kunne inkluderes i et virtuelt miljø og bli påvirket av omgivelsene på en realistisk måte.

Oppgaven gitt: 22. januar 2007
Hovedveileder: Torbjørn Hallgren, IDI

Abstract

Explosions play an important role in many games. Yet, as explosions are sudden and highly turbulent, animating visually pleasing and dynamic real-time explosions is a very complex task. We present a method for animating explosions completely on the graphics processing unit (GPU). The simulation allows for arbitrary internal boundaries and is governed by a combustion process, a Stable Fluids solver, which includes thermal expansion, and turbulence modeling. The simulation results are visualized by two particle systems comprised of textured and animated particles. The results are physically based, non-repeating, and dynamic explosions that performs in real-time with high visual quality.

Preface

This is a master's thesis for the Master of Science in Technology (Computer Science) program at the Department of Computer and Information Science (IDI). The thesis was written by Lars Andreas Ek and Rune Vistnes during our 10^{th} and final semester at the Norwegian University of Technology and Science (NTNU).

During the course of the project we have written a paper, which will be submitted to *Afrigraph 2007*. The paper is appended at the end of the thesis.

We would like to direct a big thanks to our supervisor Odd Erik Gundersen, who has motivated us, given us valuable guidance and feedback, and encouraged us to submit papers about our results and findings.

Trondheim July 23, 2007.

Lars Andreas Ek

Rune Vistnes

Contents

List of Figures

List of Tables

Chapter 1
Introduction

In this chapter the motivation behind the thesis is explained, as well as the goals it seeks to accomplish. Next, the approach we will take to fulfill them is presented. The chapter is concluded with the structure of the thesis, followed by a short summary.

1.1 Motivation

Explosions are not regular events in most humans' lives. However, thanks to the entertainment industry, most of us have a firm conviction of what they look like and how they behave. They tend to occur in entertainment media more often than not, and with the increasing demand for realistic visual quality, research has been made on explosions along with numerous other natural phenomena.

Explosions have been artificially recreated by the movie industry for quite some time. Simulations based on the underlying physical processes of explosions have produced realistic and visually convincing results.

The downside of using such methods is their computational expense. Explosions are very complex and involve highly turbulent motion that is difficult to simulate. This is not the biggest of problems within the movie industry. They do not have any real-time requirements, and thus have the luxury of performing these time consuming simulations. For computer games, however, computation complexity poses a problem. This is also reflected by the fact that explosions in most games are created using simple approximations instead of physical simulations.

The approximation methods that many games utilize often produce explosions of great visual quality, but they generally lack ability to adapt to the scene in a physically plausible fashion. Our view is that physically based simulations have great potential towards producing more realistic and interesting explosions.

In recent years, the programmability and computational power of graphics processing units (GPUs) have increased, making them useful for general purpose computation such as fluid simulations. Thus, it is interesting to investigate how GPUs can be used for physically based simulation of explosions that is suitable for real-time environments.

1.2 Objectives and requirements

Our main goal is to use a physically based simulation to visualize 3D explosions in real-time. Our focus will be on reproducing the fire ball that is often seen when explosions occur. We will also focus on letting the simulation be affected by scene geometry. More specifically, we seek to accomplish the following requirements, which eventually will be used to evaluate our results:

R1: A physically based simulation of explosions should be performed.

R2: The overall motion of the explosion should be convincing.

R3: It should be possible to place the explosion in a virtual scene.

R4: The simulation should be affected by obstacles within the scene.

R5: The visualization should be convincing, thus have realistic shape, color, and turbulence.

R6: All of the above should be performed in real-time. We require a minimum of 30 frames per second, since this is usually enough to convince the human eye.

As mentioned, one of the major advantages of real-time physically based simulation of explosions, compared to simple approximations or precalculated physically based ones, is their ability to adapt to changes in the scene. For example, a moving car that explodes under a bridge will appear differently than if the car explodes in the open. If one ignores this fact, game developers could just as well generate explosions in advance and replay the simulation in-game, hence it is an important requirement that the explosion can be affected by obstacles within the scene.

1.3 Approach

In order to achieve the requirements of real-time simulation and visualization of explosions, we will take the following steps:

- First, we will look at the physical properties of explosions, and how they may be simulated.
- Then, we will look at a variety of existing work on both offline and real-time attempts of simulating explosions, as well a related effects such as fire and smoke.
- We will develop a physically based method for simulation of explosions that can be visualized in a 3D environment. Doing so, we will attempt to combine the best methods and ideas from previous approaches.
- We will then implement the suggested method. In order to fulfill the real-time requirements we will utilize the computational power of modern GPUs, both for the simulation and the visualization.
- Next, we will test the implementation and evaluate the results.
- Finally, we will discuss the results and limitations of the method, and suggest areas for further work.

This thesis is a natural extension of the project [EV06] we carried out during the fall of 2006 in the course "TDT4715 Algorithm Construction, Science of Computing and Graphics, Specialization" at NTNU. The project compared two different methods for fluid simulation and their usefulness for physically based simulation of explosions; the Stable Fluids method [Sta99], and the Lattice Boltzmann method [WLMK04]. We concluded that the Stable Fluids method were the most suitable, mainly because of numerical instability problems with the Lattice Boltzmann method. Thus, we will use the Stable Fluids method in this thesis. Both methods were implemented in 2D using the CPU. Screen captures of the results from the Stable Fluids method are shown in figure 1.1.

1.4 Structure

This thesis contains the following eight chapters:

1 Introduction
Contains the motivation behind the thesis, presents our objectives and requirements, and the approach we intend to take to fulfill them.

2 Background
Provides the necessary theoretical background and serves as a foundation for the rest of the thesis.

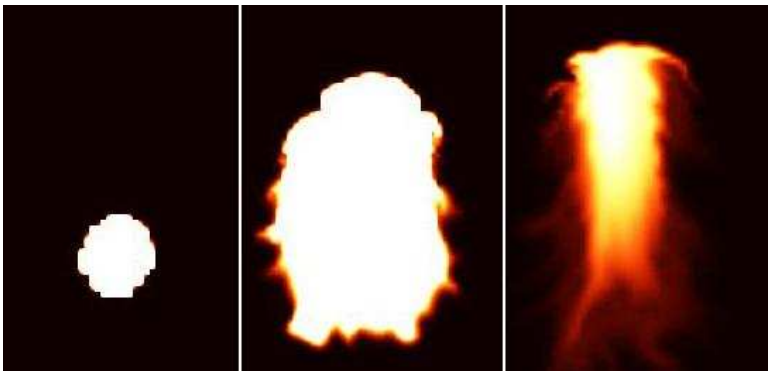

Figure 1.1: Screen captures of three different time steps of the explosions simulation based on the Stable Fluids method.

3 Related Work
Presents existing work within related areas of simulation and visualization of explosions.

4 Simulating Explosions
Presents our suggested method for simulation of explosions.

5 Visualizing Explosions
Presents our suggested method for visualizing the simulation results.

6 Implementation
Explains the implementation, and how we utilize the programmability of modern GPUs.

7 Results
Presents and discusses the results provided by the implementation.

8 Evaluation
Discusses, evaluates, and concludes the thesis, presents our main contributions and gives suggestions for future work.

1.5 Summary

Explosions are commonplace in games, but are often created using simple approximations. Such approaches generally lack the ability to be affected by scene geometry. A physically based method for simulating explosions has, in our opinion, huge potential towards producing more realistic results. Methods for physically based simulation of explosions exist, but they are generally not suited for real-time simulation. We will look at existing work

and develop a method for real-time physically based animation of explosions. The method will be implemented and tested. A set of requirements is defined to be able to evaluate the results.

Chapter 2
Background

In order to fully understand the rest of this thesis, an introduction to some of the areas and techniques that are used is required. First of all, to be able to simulate explosions, it is important to have an understanding of the physics behind them. Also, since the simulation is going to be based on fluid dynamics, an introduction to this field in addition to how fluid dynamics can implemented computationally is given. The computational power of modern GPUs is going to be utilized to meet the real-time requirements, and thus, and introduction to general-purpose computation on the GPU will be given as well.

The rest of the chapter will present each of these areas in turn. First, the physics of explosions is described. This description will focus mainly on gas and dust explosions since these often are the most visible types of explosion. Next, an overview of fluid dynamics and computational fluid dynamics will given. Both these sections are based on our previous work [EV06], but have been rewritten and adapted to better fit the domain of this thesis. Finally, an introduction to general-purpose computation on the GPU is given.

2.1 Physics of explosions

An explosion can generally be described as a rapid increase of pressure caused by a sudden increase in volume and release of energy. The pressure creates an outward-going blast wave that propagates through the surrounding medium, and causes nearby objects to accelerate outward, deform, or shatter. This blast wave is generally considered the primary effect of an explosion. However, an explosion can cause secondary effects as well. In addition to a violent release of thermal energy, noticeable visual effects include flashes of light, fire balls, whirls of dust, and flying debris [Mic92, YOH00, FOA03].

2.1.1 The combustion process

The rapid increase of pressure is in most cases caused by a chemical reaction between a combustible substance, A, and an oxidizer, B. This chemical reaction, often referred to as combustion, creates one or more combustion products, M, in addition to heat, Q, as shown by equation 2.1 [Khi62].

$$A + B = M + Q \tag{2.1}$$

The rate of this chemical reaction will vary depending on the properties of the substances. Also, initial conditions such as the current pressure and temperature will have an affect on the reaction rate. A higher reaction rate will typically cause the pressure wave to move at greater speeds than ones caused by slower reactions. Also, because the chemical reaction generates heat the reaction products will often have increased volume and pressure relative to the initial substances. This phenomenon, called thermal expansion, will cause an additional increase in the speed of the pressure wave.

The hot combustion products and the burning gas will rise upward due to buoyancy and cause a pressure drop. Because of the introduced pressure difference, colder gases usually flow in from areas underneath of higher pressure, resulting in the rolling motion often seen in explosions, as shown in figure 2.1. Here, the red arrows represents hot air, whereas the blue arrows represents colder air. The rolling motion of a real explosion can be seen in figure 2.2.

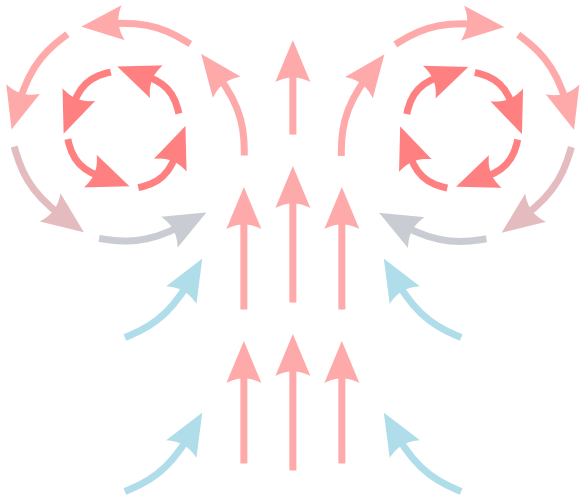

Figure 2.1: The rolling motion of an explosion

2.1.2 Classification of explosions

Explosions are categorized based on the speed of the resulting pressure wave relative to the speed of sound. Explosions whose pressure wave moves with a speed below the speed of sound (subsonic) are categorized as deflagrations, whereas explosions where the speed of the pressure wave is above the speed of sound (supersonic) are categorized as detonations. The main difference between the two types of explosions is the way they propagate outward while igniting unignited combustibles. Deflagrative combustion typically propagate by transferring thermal energy to nearby combustibles, causing them to ignite. A chain reaction of ignition and expansion takes place, provoking the formation of the pressure wave. Detonations, on the other hand, propagate simply by compressing the combustibles so that they heat up to their ignition point. Because of their powerful nature, detonations are usually the most devastating form of explosions. [Khi62, Mic92]

Figure 2.2: Rigged gasoline explosion [Wik]

2.1.3 Gas and dust explosions

Many accidental explosions are gas and dust explosions. They are commonly deflagrative but may also accelerate to reach detonative velocities. The mechanisms of generation, up-keeping, and migration of dust and gas clouds differ substantially [Eck06].

2.1.3.1 The generation of gas explosions

Gas explosions can occur when a gas or vaporizing liquid is released and ignited after the formation of a fuel-oxidiser cloud. If the gas is ignited before such a cloud has been formed, the gas will burn and not explode. Figure 2.3 shows these possible events. The leftmost box represents the release of a gas or a vaporizing liquid, from which there can be three different outcomes. The combustible may not be ignited at all, ignited immediately and result in fire, or form an explosive fuel-oxidizer cloud. This cloud is only explosive if the concentration of fuel lies between a lower and upper explosive limit. If the concentration is below the lower explosive limit, there is not enough fuel to sustain the combustion. If the concentration exceeds the explosive limit, there is not enough oxygen (or another oxidizer) to begin the reaction. Gaseous clouds sustain themselves and spread by random molecular movement and may easily migrate through very narrow passages.

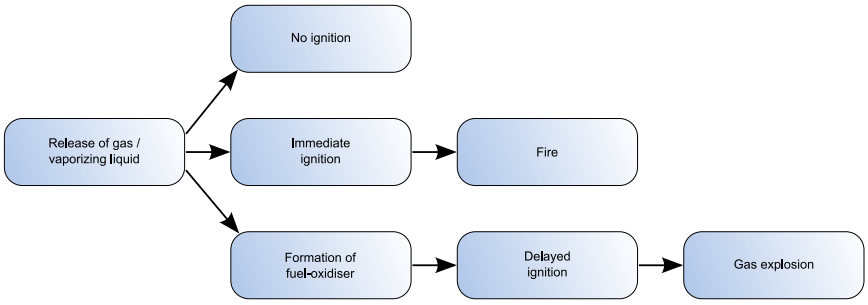

Figure 2.3: Course of events when a gas or vaporizing liquid is released

2.1.3.2 The generation of dust explosions

Dust explosions, on the other hand, can occur when small particles of a highly flammable substance, such as coal, aluminum, or gunpowder are dispersed in air and ignited [Cas00]. The particles are generally larger than gas molecules, and their movement is more affected by inertial forces, such as gravity, rather than random molecular motion. Dust clouds consisting of such particles are often sustained by interaction with moving objects, mechanical vibrations, or air-flow [Eck06].

When heat is applied near such dust clouds, the particles will vaporize, mix with the air, and become flammable. As soon as the concentration of gas exceeds the lower explosive limit, and if enough heat is present, the gas will ignite. The flame front will propagate through the mixture, causing nearby dust particles to vaporize and ignite as well [Cas00]. The applied heat that

caused the particles to vaporize will usually ignite the resulting gas before the concentration exceeds the upper explosive limit, and thus dust clouds are in practice mostly affected only by a lower explosive limit. An example of a coal dust explosion is shown in 2.4.

Figure 2.4: Staged coal-dust explosion in the Bruceton Experimental Mine [FOA03].

2.1.3.3 The conditions for gas and dust explosions

Explosive gas and dust clouds both exhibit similar ignition and combustion properties [Eck06], and they both adhere to the explosion pentagon. This pentagon, shown in figure 2.5, contains the five required conditions that have to be met if an explosion is to happen. If any of the conditions are not met, the explosion will not occur. First of all, both *fuel* and an *oxidizer* have to be present. These have to be mixed and *dispersed* or *suspended* in the air before the resulting mixture is *heated* to its ignition point. Finally, the explosive cloud have to be *confined*. This condition is often satisfied by mechanical boundaries, though an explosive cloud can also be self-contained. The latter is the case when the explosive reaction builds up in the mixture faster than it can be released in its edges [Cas00].

2.1.3.4 Heat sources

One of the conditions for an explosion is heat, and the required heat for a mixture to reach its ignition point differs from mixture to mixture. Vari-

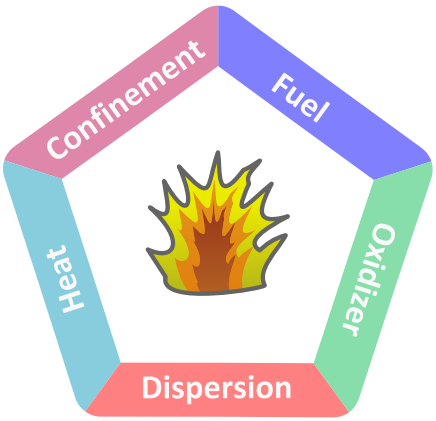

Figure 2.5: Explosion pentagon

ous heat sources can be used, and examples include open flames, cigarettes, thermal energy from controlled explosions (i.e. the mining industry), mechanical induced sparks, or electrostatic discharges between objects. The cloud may also self-ignite if its temperature is high enough [RE06].

2.1.3.5 The relation between temperature and pressure

The combustion products of gas and dust explosions are typically gases, so the process may be simplified to follow the ideal gas law:

$$PV = nRT \tag{2.2}$$

where P is the pressure, V is the volume, n is the number of moles, R is the universal gas constant, and T is the temperature. A typical accidental explosion occur in air, which contains about 70 % Nitrogen. Nitrogen will not be a part of the chemical reaction. Hence, there are usually small changes in the number of moles. From these assumptions and the use of equation 2.2, a rapid reaction in a closed system (constant volume) results in:

$$\frac{P_{\max}}{P_0} = \frac{T_b}{T_0} \tag{2.3}$$

where P_{max} is the maximum explosion pressure, P_0 is the initial pressure, T_b is the absolute temperature of the burned gas and T_0 is the absolute initial temperature [Cas00]. The faster the reaction rate, the explosion is more likely to follow equation 2.3, since less thermal energy will leave the system.

2.1.3.6 The effect of turbulence on the reaction velocity

Also common to gas and dust explosions is the effect of turbulence on reaction velocity. Turbulence is a irregular random fluctuation in the flow, as opposed to non-turbulent or laminar flow [Mic92] and it greatly increases the surface between hot burning gas and the unignited gas. The increased area causes turbulent flames to reach velocities much greater than those of laminar flames, because more gas may be ignited simultaneously. The leftmost illustration of figure 2.6 shows a laminar boundary between an area of burning gas and unignited gas, while the rightmost illustration shows a turbulent boundary. Clearly, the rightmost has a greater surface.

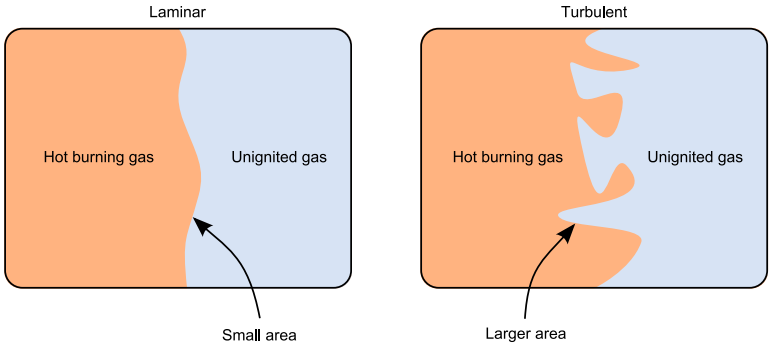

Figure 2.6: Laminar (left) and turbulent (right) flame propagation

2.2 Fluid dynamics

The study of fluids in motion, generally known as fluid dynamics, can be used to physically model many natural phenomena, including smoke, fire, and explosions. The use of fluid dynamics to model such phenomena require a few basic assumptions however. First, it has to be assumed that the mass is conserved. That is, the rate of mass passing from inside to outside of a surface must be the same as the rate of mass passing from outside to inside. Note that the rate of mass passing can be zero. Also, the total momentum, that is, the sum of the products of the mass and velocity of objects, has to be conserved.

2.2.1 The Navier-Stokes Equations

The conservation laws of mass and momentum are used to derive a set of equations known as the Navier-Stokes equations, which can be used to

describe the motion of fluid. The detailed steps resulting in these equation are outside the scope of this thesis, and we refer the reader to [GDN97] and [Cra04] for further details. Instead, the Navier-Stokes equations for compressible viscous flow are presented along with possible assumptions for simplifying them.

The Navier-Stokes equations for the conservation of momentum are are:

$$\frac{\delta(\rho\mathbf{u})}{\delta t} + (\mathbf{u}\cdot\nabla)(\rho\mathbf{u}) + (\rho\mathbf{u})(\nabla\cdot\mathbf{u}) + \nabla p = (\mu+\lambda)\nabla(\nabla\cdot\mathbf{u}) + \mu\nabla^2\mathbf{u} + \rho\mathbf{f} \quad (2.4)$$

$$\frac{\delta\rho}{\delta t} + \nabla\cdot(\rho\mathbf{u}) = 0 \quad (2.5)$$

Here, equation 2.4 is the momentum equation that describes how the momentum changes over time, while equation 2.5 gives a restriction that ensures the conservation of mass. In these equation $\mathbf{u}$ is the velocity, ρ the density, and p is pressure. μ and λ are scalar values specific to the fluid, while $\mathbf{f}$ represents external forces.

In addition to boundary and initial conditions, to solve these equations we need a function relating pressure and density:

$$p = f(\rho)$$

Also, as a further complication, the μ and λ scalars are generally not constants, and usually vary based on the density and temperature of the fluid:

$$\mu = \mu(T,\rho)$$
$$\lambda = \lambda(T,\rho)$$

2.2.2 Simplifying assumptions

Equations 2.4 and 2.5 are complex. However, when modeling fluids for a specific case, knowledge of the context can be used to justify assumptions that will simplify these equation. Two cases where such simplifications can be made are when modeling incompressible fluids instead of compressible ones, and when the fluid is inviscous instead of viscous.

2.2.2.1 Incompressible fluid

A common simplification is to assume that the fluid is incompressible. A fluid is incompressible as apposed to compressible whenever density variations are so small they can be neglected. In other words, constant density is assumed. Applying this assumption to equations 2.4 and 2.5 results in equations 2.6 and 2.7 respectively:

$$\rho\frac{\delta\mathbf{u}}{\delta t} + \rho\left(\mathbf{u}\cdot\nabla\right)\mathbf{u} + \nabla p = \mu\nabla^2\mathbf{u} + \rho\mathbf{f} \tag{2.6}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{2.7}$$

The mass conservation restriction simplifies to ensuring zero divergence (equation 2.7) and is subsequently used to simplify the momentum equation. As can be seen, there is no longer a need for the scalar value λ, or the relation between the pressure, p, and the density, ρ.

The incompressible version of the momentum equation, equation 2.6, can be more easily interpreted if divided by ρ and presented with the velocity differential on the left side and all the other terms moved to the eight side of the equal sign. Note that a viscosity factor v is introduces instead of $\frac{\mu}{\rho}$:

$$\frac{\delta\mathbf{u}}{\delta t} = -\left(\mathbf{u}\cdot\nabla\right)\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2\mathbf{u} + \mathbf{f} \tag{2.8}$$

The first term in equation 2.8, $-\left(\mathbf{u}\cdot\nabla\right)\mathbf{u}$, is called the advection term and describes how the momentum of the fluid is carried along by itself. The second term, $-\frac{1}{\rho}\nabla p$, represents the fluid's tendency to remove pressure differences, causing motion from areas of high pressure to areas of lower pressure. The third term, $v\nabla^2\mathbf{u}$, models the viscosity of the fluid. Viscosity causes velocity to spread, and can be thought of as the fluid's resistance to motion. For instance, syrup has a high resistance to motion, while air has not. Finally, the last term, $\mathbf{f}$, represents the effect of external forces on the fluid.

Even though the incompressible version of the Navier-Stokes equations is less complex than the compressible one, it can only be used in cases where the assumption of incompressibility can be justified. Liquids may often be modeled as incompressible, as they tend to be difficult to compress. Gases, on the other hand, should generally be modeled as compressible. However, there are circumstances where an incompressible model can be used instead. A rule of thumb is to use a compressible model if the fluid velocities exceed a Mach number of 0.3 [KKS99], and an incompressible model otherwise. Such

a relation between velocity and constant density is rather intuitive, because a slowly moving fluid has the time to eliminate density differences before they become relevant.

2.2.2.2 Inviscous fluid

Another useful simplification is to assume that the effect of the viscosity can be neglected. By setting the viscosity factor, v, to zero, the viscosity term of equation 2.8, $v\nabla^2\mathbf{u}$, will be ignored.

As with the incompressibility assumption, fluids can only be treated as inviscous in cases where such an assumption can be justified. Typically, an inviscious model can be used for fluids with low viscosity, since the viscosity have a limited affect on the motion of such fluids. For instance, the viscosity in syrup is too high to be ignored, while gases has a lower viscosity and can thus often be modeled with an inviscous model.

Even though an inviscous model may not always be suitable, and an viscous model have to be used instead, it can often be assumed that the viscosity factor, v, does not vary with density and temperature.

2.2.3 Computational fluid dynamics

Computational Fluid Dynamics (CFD) is the use of numerical methods to analyze and solve problems in fluid dynamics. Computers and specialized algorithms are used and operate in a spatial domain that is usually discretized into a grid of small cells. Each cell represents the state of a small part of the fluid, such as the density, the velocity, and the temperature. These values are then updated in discrete timesteps.

The Navier-Stokes equations are used as the fundamental basis in CFD calculations. Many different algorithms exists, and common to many of them is that they only approximate the solutions instead of accurately solving them. It is also common to assume that the fluid is either incompressible or inviscous, or both, as previously described. However, many of the algorithms suffer from stability issues, especially with larger timesteps.

2.2.4 The Stable Fluids method

One approach that approximates the incompressible viscous Navier-Stokes equations is the one presented by Stam in [Sta99]. This method is unconditionally stable, and have thus become popular and heavily adopted, es-

pecially in real-time simulations, which often require larger timesteps than the ones possible with more unstable models. The mathematics behind the method is briefly explained here, and the reader is referred to [Sta99] and [Har04] for more thorough explanations.

2.2.4.1 Mathematical background

As already mentioned, Stam's approach makes use of the incompressible viscous Navier-Stokes equations 2.8 and 2.7. These equations are repeated for the ease of readability:

$$\frac{\delta \mathbf{u}}{\delta t} = -(\mathbf{u} \cdot \nabla)\,\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2 \mathbf{u} + \mathbf{f} \tag{2.8}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.7}$$

Normally, in order to solve the equations as is, an explicit pressure field needs to be available. This pressure field is used when evaluating the pressure term, $-\frac{1}{\rho}\nabla p$. However, this pressure term can be removed by the use of a projection operator, $\mathcal{P}$, introduced by Stam on both sides of equation 2.8, as shown in equation 2.9:

$$\mathcal{P}\left(\frac{\delta \mathbf{u}}{\delta t}\right) = \mathcal{P}\left(-(\mathbf{u} \cdot \nabla)\,\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2 \mathbf{u} + \mathbf{f}\right) \tag{2.9}$$

The projection operator is defined to remove the divergence from any vector it is applied to by projecting the vector onto it's divergence free component. Mathematically, this can be explained by the use *Helmholtz-Hodge Decomposition* [CM93] that states that any vector field, $\mathbf{w}$, can be uniquely decomposed into the form:

$$\mathbf{w} = \mathbf{v} + \nabla q \tag{2.10}$$

where $\mathbf{v}$ is a divergence free vector field and q is a scalar field. ∇q is thus the divergence of $\mathbf{w}$, and applying the P-operator to $\mathbf{w}$ will result in $\mathbf{v}$:

$$P(\mathbf{w}) = \mathbf{w} - \mathbf{w}_{divergence} = (\mathbf{v} + \nabla q) - \nabla q = \mathbf{v}$$

Since the velocity is defined to be divergence free, equation 2.9 can be simplified to:

$$\frac{\delta \mathbf{u}}{\delta t} = P\left(-\left(\mathbf{u}\cdot\nabla\right)\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2\mathbf{u} + \mathbf{f}\right) \tag{2.11}$$

Next, to see what happens to the pressure term when the projection operator is applied to it, imagine that equation 2.10 is multiplied by the ∇-operator. Since $\mathbf{v}$ is divergence free, $\nabla\cdot\mathbf{v} = 0$, this will result in equation 2.12 [1]

$$\nabla\cdot\mathbf{w} = \nabla^2 q \tag{2.12}$$

Here, $\mathbf{w}$ is a vector field, and ∇q is it's divergence. This equation is often referred to as the Poisson-pressure equation. Now, if $\mathbf{w}$ is set to be the gradient of p, ∇p, then clearly p equals q, shown as follows:

$$\nabla\cdot(\nabla p) = \nabla^2 q \rightarrow \nabla^2 p = \nabla^2 q \rightarrow p = q$$

Since ∇q is the divergence of $\mathbf{w}$, ∇p is the divergence of ∇p, and thus, the result of applying the $\mathcal{P}$-operator on the pressure term is zero:

$$P\left(\frac{1}{\rho}\nabla p\right) = \frac{1}{\rho}P\left(\nabla p\right) = \frac{1}{\rho}\left(\nabla p - \nabla p\right) = 0$$

Thus, equation 2.11 can be simplified to equation 2.13.

$$\frac{\delta \mathbf{u}}{\delta t} = P\left(-\left(\mathbf{u}\cdot\nabla\right)\mathbf{u} + v\nabla^2\mathbf{u} + \mathbf{f}\right) \tag{2.13}$$

2.2.4.2 Simulating velocities

Equation 2.13 shows the steps of the Stable Fluids algorithm. First the advection, viscocity, and force terms are calculated and summarized. The resulting velocity field is divergent, and is made divergence free by the $\mathcal{P}$-operator.

In practice, the three terms are not calculated and summarized as in equation 2.13. Instead, each term is accounted for on it's own, each producing a new velocity field with the velocity field from the prior step is input. This can be described by the use of an introduced $\mathcal{S}$-operator, whose result is equivalent to the calculation of equation 2.13 over one time step:

[1] $$\nabla\cdot\mathbf{w} = \nabla\cdot(\mathbf{v}+\nabla q) = \nabla\cdot\mathbf{v} + \nabla\cdot(\nabla q) = \nabla^2 q$$

$$\mathcal{S}(\mathbf{u}) = \mathcal{P} \circ \mathcal{A} \circ \mathcal{V} \circ \mathcal{F}(\mathbf{u}) \tag{2.14}$$

Here, $\mathcal{P}$ is the projection operator, $\mathcal{A}$ the advection step, $\mathcal{V}$ the viscosity step, and $\mathcal{F}$ the force step. The four steps are applied from right to left. First, the forces are added, followed by the viscosity step, the advection step, and finally the P-operator.

Adding external forces
The first, and also the easiest step, is to add external forces. These forces typically come from buoyance, gravity, and wind. The forces are assumed to remain constant during each time step Δt, thus they can simply be added as shown by equation 2.15.

$$\mathbf{u}^{new}(x) = \mathbf{u}(x) + \Delta t \mathbf{f}(x) \tag{2.15}$$

Here, $\mathbf{f}$ is a vector field of forces, $\mathbf{u}$ is the current velocity field, and $\mathbf{u}^{new}$ is the resulting velocity field that is used as input to the viscosity step.

Viscosity
The next step is the viscosity step. Note that if the fluid is assumed to be inviscous, this step can be skipped. Instead of using explicit finite difference derivation, which is unstable for large time steps [FM97], Stam proposed to use an implicit integration scheme, as shown by equation 2.16:

$$\left(\mathbf{I} - v\Delta t \nabla^2\right) \mathbf{u}^{new}(x) = \mathbf{u}(x) \tag{2.16}$$

Here, $\mathbf{u}$ is the velocity field from the force step, $\mathbf{u}^{new}$ is the resulting velocity field that is used as input to the advection step, v is the viscosity factor, and $\mathbf{I}$ is the identity matrix. The equation yields a system of linear equations, which can be solved by an iterative linear equation solver, such as the Jacobi or Gauss Seidel method.

Advection
The third step is the advection step. As with the viscosity step, an explicit integration with finite difference derivation can be used. However, because of the potential stability, Stam proposed a stable method where he regarded each cell as a particle and used the velocity at the particle's current position to locate where it must have been at the previous time step. The new velocity is then set to the velocity at the position at the previous time step. Since this position can be located in between cells, the value is calculated by bilinearly interpolating the velocity value from the four most nearby cells. This can be expressed mathematically by equation 2.17:

$$\mathbf{u}^{new}(x) = \mathbf{u}(x - \Delta t \mathbf{u}(x)) \tag{2.17}$$

where $\mathbf{u}$ is the velocity field from the viscosity step and $\mathbf{u}^{new}$ is the resulting velocity field that is used as input to the projection step.

Figure 2.7 shows how a particle at $\mathbf{u}(x,t)$ is traced back in time over a time step Δt to find its previous position. It also shows the four cells whose values are interpolated to find the new velocity.

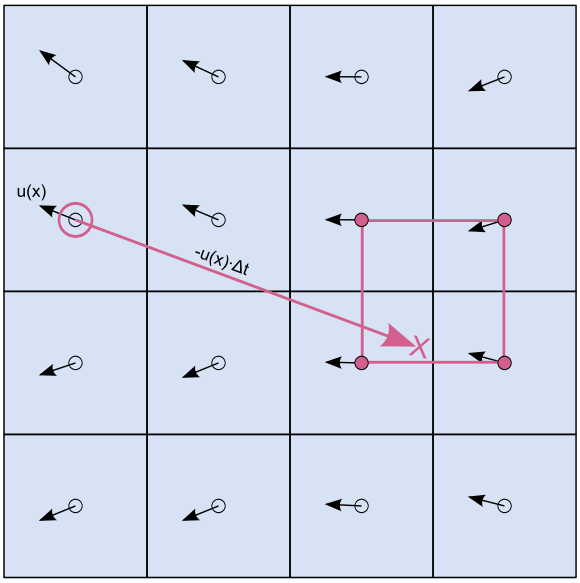

Figure 2.7: Advection (The figure is inspired by a figure in [Har04]).

Projection

The final step is the projection step. Since the velocity field have been made divergent by the three prior steps, the projection operator is applied in this step to remove the divergence. To do so, equation 2.12 is used, repeated here for the ease of readability:

$$\nabla \cdot \mathbf{w} = \nabla^2 q \tag{2.12}$$

$\mathbf{w}$ is set to be the divergent velocity field, and the equation is solved for q, whose gradient is then subtracted from the velocity field, resulting in a divergence free velocity field. As with the viscous diffusion equation, the Poisson equation can also be solved by the use of an iterative linear equation solver.

2.2.4.3 Simulating densities

In addition to showing how to solve the momentum equation, Stam proposed how to introduce densities, such as the temperature, to the simulation. The densities will be carried along by the velocity while being affected by molecular diffusion. Stam also let the densities dissipate at a constant rate and included a "force term" to allow densities to be added to the simulation. Equation 2.18 shows the evolving of the densities from one time step to the next:

$$\frac{\delta d}{\delta t} = -\left(\mathbf{u} \cdot \nabla\right) d + \kappa_d \nabla^2 d - \alpha_d d + S_d \tag{2.18}$$

where d is the density, $\mathbf{u}$ is the velocity field, κ_d is the diffusion rate, α_d is the dissipation rate, and S_d is the source term. The terms in equation 2.18 are very similar to the ones in equation 2.13 and may in fact be solved using the same methods.

Sources
The source term may be considered as a constant force affecting the density during each time step, hence it may be treated exactly as the force term in equation 2.13. Similarly, the dissipation is also considered to be constant, and the two may be combined in equation 2.19:

$$d^{new} = d + \Delta t \left(S_d - \alpha_d d\right) \tag{2.19}$$

Here, d is the old density field and d^{new} is the new density field, which is used as input to the advection step.

Diffusion
The diffusion term is solved implicitly as with the viscosity step as shown by equation 2.20.

$$\left(1 - \kappa_d \Delta t \nabla^2\right) d^{new}\left(x\right) = d\left(x\right) \tag{2.20}$$

Advection
The advection term is solved using the exactly same method as for the velocity field, except the density is carried along, not the velocity itself.

2.3 General-purpose Computation using Graphics Hardware

Graphics hardware, generally in form of a graphics processing unit (GPU), have in the later years been used for various computer graphics tasks such as games. These tasks typically have high demands in performance and visual quality, and in order to meet these demands, the architecture on which GPUs are built has been highly specialized during years of evolving and tuning.

One of the main contributors to the high performance of GPUs is the highly parallel nature of the data used in most computer graphic tasks. Parallelism is introduced into GPUs by enabling it to perform computations on several streams of data simultaneously. GPUs are also optimized for vector operations, resulting in an even greater performance increase.

An increased flexibility has also been introduced in order to give the users of the GPU better control of the tasks they want to perform. This increased flexibility has enabled the use of GPUs in applications outside the scope of which the GPUs were originally designed for. These applications typically have to be adopted into computer graphics terms. Even though this adoption is often challenging, and sometimes impossible due to the nature of the problem and the limitations of graphics hardware, the potential benefits in utilizing graphics hardware instead of the CPU for general-purpose problems may be substantial.

2.3.1 The pipeline

To allow high computation rates through parallel execution, most of the graphics hardware today organize their computation in a similar fashion, known as the graphics pipeline. Conceptually, as seen in figure 2.8, this pipeline is divided into three functional stages: the application stage, the geometry stage, and the rasterization stage [AMH02]. First, the application issues drawing commands and outputs 3D primitives representing the scene. Next, these 3D primitives are transformed into 2D primitives within the geometry stage by projecting them into the screen field-of-view. Finally, the rasterization stage fragments the 2D triangles into pixels and computes a color for each pixel.

As shown in figure 2.8, a conceptual stage may contain substages and thus be a pipeline in itself. These substages may also be partly parallelized, as shown in the rasterizer stage. As opposed to the three conceptual stages, these substages can be thought of as either functional stages or pipeline stages. A functional stage has a certain task to perform, but does not specify

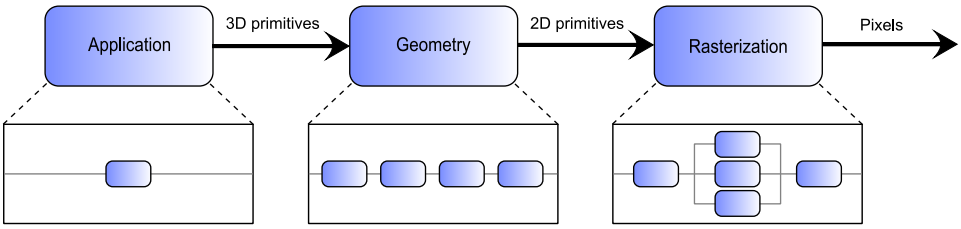

Figure 2.8: A conceptual overview of the graphics pipeline

the way it is executed in the pipeline. A pipeline stage, on the other hand, is the actual implementation of the pipeline. For example, the geometry stage may be divided into five functional stages, but it is the implementation of the graphics system that determines its division into pipeline stages. A given implementation may combine two functional stages into two pipeline stages, while it divides another, more time-consuming, functional stage into several pipeline stages, or even parallelizes it. [AMH02]

The following sections will describe each of the three conceptual stages in more detail. The focus will be on the functional substages only, and not the actual pipeline stages.

2.3.1.1 The application stage

The application stage is the first stage in the graphics pipeline, and differs from the other stages by the fact that it is normally performed on the CPU instead of the GPU. Using a graphics framework such as OpenGL or Direct3D, the application sends the geometry that is to be rendered along to the next stage in the pipeline. This geometry consists of several 3D primitives, such as points, lines, and triangles. The primitives are defined by one or more vertices, and these vertices are given various attributes, such as a position, color, normal, or a texture coordinate. These attributes are used by later stages when positioning and coloring the primitives.

Examples of other common processes often implemented in this stage include collision detection, input handling, and artificial intelligence. These stages are hard or impossible to implement efficiently in later stages. Also, since the performance of the next stages in the pipeline depends on the number of primitives being processed, various techniques are used in the application stage to minimize the amount of primitives sent to the geometry stage. Since often only parts of a scene are visible at a given time, this typically includes techniques to detect the primitives that is guaranteed to be culled in the geometry stage, and not send these to the geometry stage in the first place.

2.3.1.2 The geometry stage

The geometry stage is the first step performed on the GPU. It takes the 3D vertices from the application stage as input, and it's main objective is to transform these 3D primitives into 2D primitives that will be passed on to the rasterization stage.

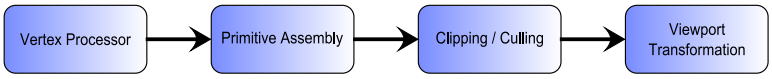

Figure 2.9: The functional stages of the geometry stage

In order to perform this transformation, the vertices are sent through several minor stages, as shown in figure 2.9. These are the vertex processor, the primitive assembly, clipping and culling, and viewport transformation. First, the vertices are processed by the vertex processor. Normally, when entering the geometry stage, the vertex positions are all declared in the object coordinate system. The vertices are then transformed into the world coordinate system by translation, scaling, and rotation. Figure 2.10 shows an example where two primitives defined in their own object coordinate system are transformed into a common world coordinate system.

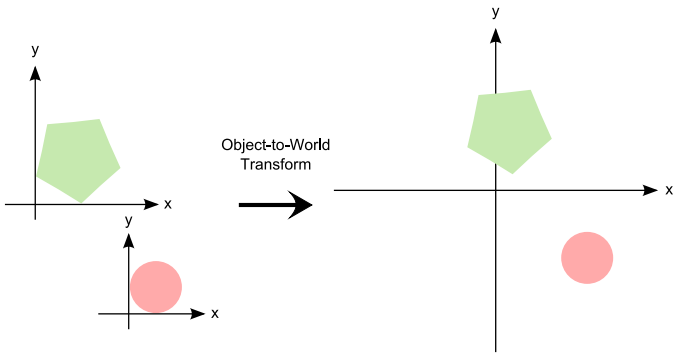

Figure 2.10: Transform from the object to the world coordinate system

Next, the vertices are transformed into the view coordinate system based on the projection and position of the viewport being used in the scene. The viewport is defined using a camera that are also positioned in the world coordinate system. Figure 2.11 shows how the primitives from figure 2.10 are transformed to the viewport defined by the camera.

The vertex processor can also calculate vertex lighting based on the position and intensity of the light sources, the eye position, the vertex normal, and the vertex color.

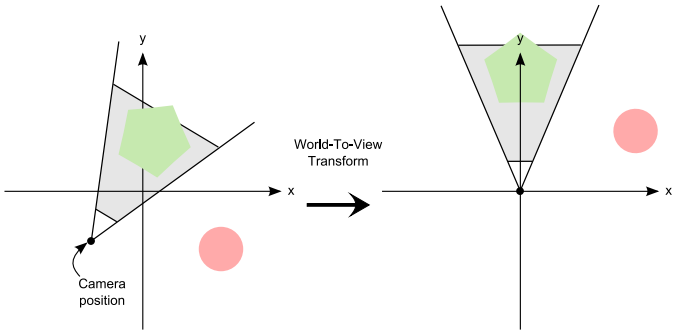

Figure 2.11: Transform from the world to the view coordinate system

The next stage in the geometry stage is the primitive assembly stage, where the vertices are assembled back into triangles using the information sent from the application stage. These triangles are then checked against the viewing frustrum defined by the camera. Primitives that are totally outside the view volume are discarded, whereas primitives that are totally inside the view volume are passed on as is. Primitives that only partially inside the view volume require clipping. Here, the vertices that are located outside the view volume are replaced with vertices that are located at the intersection between the primitive and the view volume. Figure 2.12 shows each of these three possible scenarios. Here, the red triangle is discarded, the blue triangle is passed on as is, while the red triangle is clipped, resulting in two new vertices.

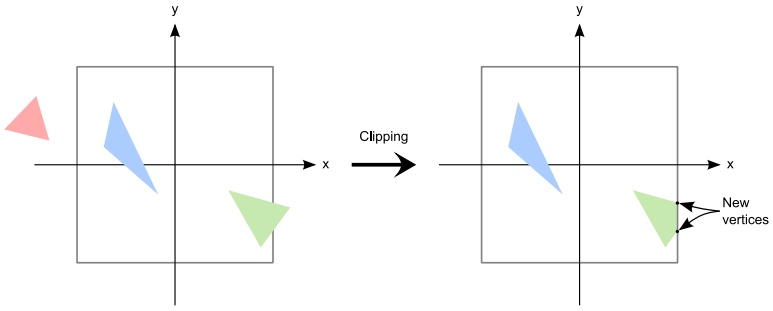

Figure 2.12: Clipping

In addition to the primitives that are totally outside the view volume, primitives that are facing away from the camera will be discarded as well, regardless of their position relative to the view volume.

Finally, in the viewport transformation stage, the vertices are transformed into the screen space domain. Here, the vertices are still located in the 3-

dimensional space. The final 2D position of the vertices on the screen is calculated, and sent along with the depth information to the rasterization stage.

2.3.1.3 The rasterization stage

The final conceptual step is the rasterization stage, where the transformed 2D primitives are converted into colored screen pixels. The rasterization stage consists of several minor stages, as seen in figure 2.13. These are the rasterizer, the pixel processor, the raster operation, and the frame buffer.

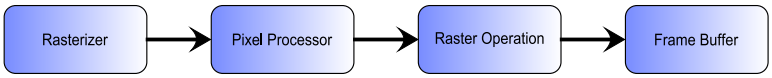

Figure 2.13: The functional stages of the rasterization stage

The main function of the rasterizer is to determine what pixels every 2D primitive consist of. Figure 2.14 shows an example of which pixels the rasterizer would determine the given triangle would consist of. It also interpolates the vertex attributes into these pixels. Every pixel is then passed along to the pixel processor that, by using these interpolated values along with other information such as the location of the pixels in the frame buffer, is able to determine the final color of every pixel. The colors of the pixels in a primitive are often based on texture lookups. The interpolated texture coordinate associated with the vertices that represent the given primitive are used to decide which texture value to be used for the given pixel.

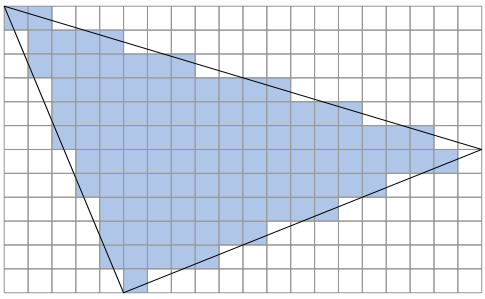

Figure 2.14: A rasterized triangle.

Since triangles often overlap, more than one pixel can be located in the same position in the frame buffer. These pixels are combined during the raster operation by the use of alpha, depth, and stencil tests before a final color is written to the frame buffer.

2.3.2 Towards general-purpose computation

Initially, every stage in the graphics pipeline were fixed, and the rather limited functionality at each stage were hardwired. The architecture was specialized for real-time computer graphics, and the included functionality was well suited for the intended field of usage. However, offline rendering systems such as Pixar's RenderMan demonstrated the benefits of increased flexibility. Using user-defined shader programs on each primitive, they were able to achieve impressive visual results that would otherwise be hard or impossible to achieve using a fixed-function pipeline.

Results such as these have motivated designers of graphics hardware to transform the fixed-function pipeline into a more flexible and programmable pipeline. Initially, two of the stages in the pipeline were programmable; the vertex processor and the pixel processor. With the introduction of DirectX 10 hardware, the primitive assembly stage became partially programmable, too, allowing dynamic generation of new primitives directly on the GPU.

The introduced programmability was first available through the use of assembly instructions. However, programs written in assembly are hard to write, read, and debug, hence it didn't take long before higher level shader languages appeared. Examples of such shader languages are OpenGL Shading Language (GLSL) and Microsoft's High Level Shading Language (HLSL). These languages typically offer a C-like syntax, making programs easier to develop. Programs written using these languages are compiled down to assembly instructions before being sent to the GPU, thus they should be comparable in speed to programs written in assembly from scratch.

This programmability along with the high speed and low cost of recent graphics hardware has become a vital step for general-purpose computation on the GPU. Even though still only intended to be used for computer graphics tasks, the programming model that was introduced is, even though unusual, general enough to be used for a large class of non-graphics problems. Also, each new generation of GPUs has increased functionality and generality, enabling the use of the GPU for an even larger class of non-graphics problems. Still, not every problem is suitable to solve using the GPU. The high speeds offered by graphics hardware are possible due to data parallelism and a highly specialized architecture. Non-graphics problem domains have to be fitted into this highly limited domain in order to take advantage of the potential speed-ups offered by graphics hardware. However, even though often challenging, the potential benefits in utilizing graphics hardware for general-purpose problems may be substantial.

2.3.3 Mapping to the GPU

A typical program that more often than not can be moved from the CPU to the GPU, is the one where most of the calculation is performed on one or more arrays of data. On the GPU, the analogy to arrays are textures. Textures can contain up to four color channels, and thus, if more that one array of data are to be used, these can be packed into the various color channels of a texture. This way, increased parallelism can be achieved since computations can be performed on all the four color channels at once.

When doing calculations on an array on the CPU, the same operation is usually performed on larger parts of or maybe even on the whole array. Loops are used to iterate over the elements in the array, and the operation is then performed on the given element in each iteration. On the GPU, these operations are represented by pixel shader programs. In order to invoke these pixel shader programs, instead of using loops, a stream of pixels has to be generated. Typically, this is done by drawing a quad parallel to the viewport, and with the viewport sized to fit the desired output array. The rasterizer then generates the pixels that are covered by the quad, and since the size of the viewport matches the size of the array, the number of pixels generated is equal the number of elements in the array. Also, the vertices that defines the quad are given a set of texture coordinates, which are interpolated into every pixel by the rasterizer. These texture coordinates are sent as input to the pixel shader programs, and they can be used to the fetch the value at the position in the texture that corresponds to the position of the pixel relative to the quad.

2.3.4 Limitations and difficulties

As already mentioned, there are several limitations and difficulties involved when programming on the GPU, and non-graphical problems in particular. Such problems can often not be solved intuitively on the GPU, and have to be translated into the graphics domain before it can be solved. This often require a deep understanding of both the graphics domain and the problem domain.

Another limitation is that it is only possible to read from and write to a texture and not the entire memory in general. Also, even though it is possible to read from an arbitrarty texture location, the position in the texture to which a calculated value is to be written is already determined before a call to an instance of a pixel shader program is made. Thus, in order to write to specific positions in a texture, the graphics pipeline have to be set up properly so that the rasterized pixels corresponds to the desired positions in

the texture. The pixel shader programs that are executed on each rasterized pixel will then be used to calculate new values at the corresponding pixels in the texture.

Textures are also limited by the fact that it is not possible to read from and write to the same texture at the same time. Instead, a temporary texture has to be written to instead, and then the two textures are swapped once the rendering operation is complete.

Yet another limitation is the lack of 64 bit double precision floating point numbers. Since some large-scale problems often require the extra precision offered by these numbers, this limitation hampers or prevents the use of the GPU for these problems.

2.4 Summary

An explosion can be described as a process that causes a sudden increase of pressure. The primary effect of this process is an outward-going pressure wave. Secondary and more visible effects occur as well, and include flashes of light, fire balls, whirls of dust, and flying debris. Explosions are categorized based on the velocity of the pressure wave. If the pressure wave has subsonic velocities, the explosion is considered a deflagration, whereas a detonation causes a pressure wave with supersonic velocities. Two types of explosions that often are highly visible are gas and dust explosions.

The motion of hot gases may be described by a set of equations known as the Navier-Stokes equations, and can thus be modeled by the use of fluid dynamics. Since the compressible versions of the Navier-Stokes equations are too time-consuming to solve in real-time, several simplifying assumptions are made. Two such simplifications can be made by assuming that the fluid is incompressible and inviscous.

Graphics hardware have a highly specialized architecture designed to meet the performance and visual quality requirements of various computer graphics tasks. The high speed is made possible by the organization of the computations into a graphics pipeline. Also, graphics hardware have in the more recent years become fully programmable. This increased programmability have not only made general-purpose computations possible, but also very attractive due to the high performance delivered by graphics hardware.

Chapter 3
Related Work

This chapter presents previous work that is related to our goal of simulating and visualizing explosions in real-time. The presentation is divided in two. First, work related to the simulation of explosions is presented, followed by a presentation of work related to the visualization of explosions. Instead of focusing on the visualization of explosions alone, various ways that volumetric effects in general can be visualized will be described instead. Examples include using particle systems as well as ray casting.

The rest of this chapter presents the previous work related to both simulation and visualization of explosions. Next, the various approaches are compared, before a short summary concludes the chapter.

3.1 Simulation of explosions

Before presenting previous work on simulation of explosions, previous work on computational fluid simulation will be presented first. There are two reasons for this. First, as we want to base our simulation on the field of computational fluid dynamics, an introduction to related work in this field is just as important as related work on explosion simulations. Second, fluid dynamics has been used in most of the explosion simulations presented section 3.1.2. Hence, in order to fully understand these approaches, an understanding of the approaches to computational fluid dynamics is important.

Previous work on simulation of explosions are presented next. Only approaches that focus on the visual effects of the explosion rather than on the pressure wave will be included, since we are not interested in modeling the pressure wave anyways.

The presentation continues by describing various ways combustion has been modeled. Combustion is not only limited to explosions, but is also present in other phenomena such as fire, and thus, the presentation will present a more general overview of work related to combustion modeling.

Two other aspects that are important to investigate are volumetric extrusion, and vorticity and turbulence modeling. Volumetric extrusion is used to approximate 3d simulations by performing the simulation in 2d and then use interpolation to move the density and velocity values into the 3d domain. Vorticity and turbulence modeling on the other hand is used to enhance the existing vorticity, and also to introduce additional turbulence.

Finally, methods that include arbitrary boundary conditions to fluid simulations are discussed.

3.1.1 Computational fluid simulation

[FM97] popularized the field of computational fluid dynamics in the graphics community with a model for simulating the motion of a hot, turbulent gas in a full 3D environment. The flow of the gas was described by the inclusion of the incompressible Navier-Stokes equations. The non-linear and differential nature of these equations has historically made them difficult to solve, and in this work an explicit integration scheme were used to solve them. The stability of this integration scheme required very small time steps, and thus put a strong limitation on the usefulness of the model in a real-time simulation. An example how they simulated hot smoke using their method is shown in figure 3.1.

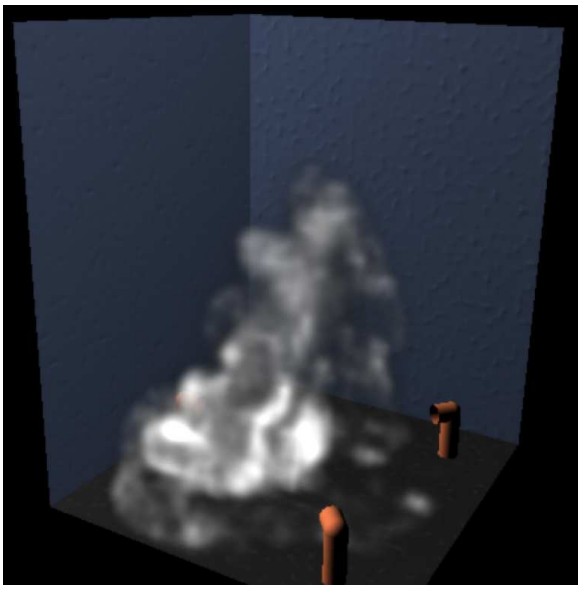

Figure 3.1: Simulation of hot smoke [FM97]

This limitation was later removed by Stam's work in [Sta99] where he pre-

sented his unconditionally stable method. Here, the solutions to the incompressible Navier-Stokes equations are approximated instead of being solved explicitly. The advection term of the incompressible Navier-Stokes equation is solved using a semi-laplacian step. Instead of moving the grid values forward along the current velocity field, new grid values are found by moving backwards in time along the velocity field to find where the value in a cell would have come from. The diffusion term is solved implicitly using an iterative linear equation solver, whereas the force term is explicitly added. The resulting vector field is divergent, and it is made non-divergent by subtracting the gradient of a pressure field. This pressure field is calculated using an iterative linear equation solver, similar to the one used to solve the diffusion term. One of the drawbacks of Stam's method is the introduction of numerical dissipation, causing unnatural damping and the loss of vorticity. Also, the linear equation solvers often requires several iterations to give satisfactory results.

3.1.2 Simulation of explosions

[FOA03] animated visually pleasing suspended particle explosions. Instead of modeling the blast wave, they modeled the motion of air and hot gases around an explosion. This was done by solving the incompressible Navier-Stokes equations in coherence with a system of fuel and soot particles. The particles were advected based on the underlying velocity field, and combustion of the particles affected the temperature and momentum. The Stable Fluids method [Sta99] was used to solve the incompressible Navier-Stokes equations. Also, they approximated compressibility by letting the divergence be zero except where mass was added or the fluid was expanded by heat. Figure 3.2 shows various time steps from an explosion simulated using their technique.

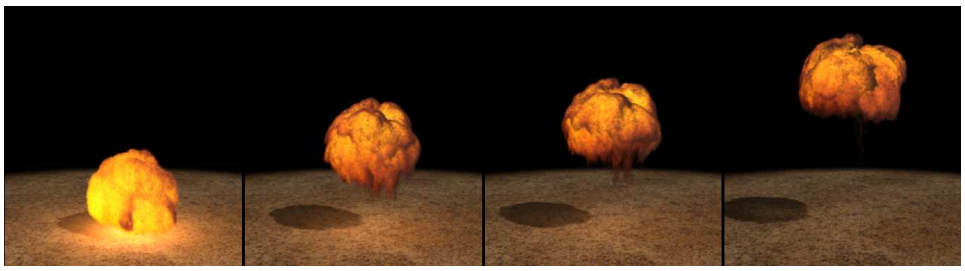

Figure 3.2: Four time steps from a suspended particle explosion [FOA03]

[RNGF03] simulated highly detailed large scale explosions using a few two-dimensional high resolution physically based flow fields. Fields existed for

both the velocity, density, and temperature of the fluid, and they were calculated by the use of the Stable Fluids method [Sta99]. These two-dimensional fields were moved into the tree-dimensional domain by the use of volumetric extrusion. They also used non-interacting particles whose motion were affected by the flow field. The velocity, density, and temperature of each particle were defined by interpolating between the two-dimensional fields. Because of the simplifications that were made, very large grid sizes could be used, and thus large scale simulations could be performed, such as the one shown in figure 3.3.

Figure 3.3: Two different large scale explosions [RNGF03]

[TOT+03] used a discrete Lagrangian fluid model in coherence with flame and air particles. The various physical quantities used in the simulation, such as the buoyancy and the pressure gradient, are calculated based on the interaction between the particles. Figure 3.4.

[KW05] used non-physical pressure and velocity templates to affect a Navier-Stokes based fluid simulation. The pressure template are used to adjust the pressure field, whereas the velocity template are used to adjust the velocity field. Using these templates, custom fluid flow can be designed as desired. Explosion effects are created by inserting a very high temperature at the desired explosion center. Buoyancy affects the velocity field, which is used to transport the heat. Torus shaped pressure templates are used to create the rolling motion of the explosion, as shown in figure 3.5.

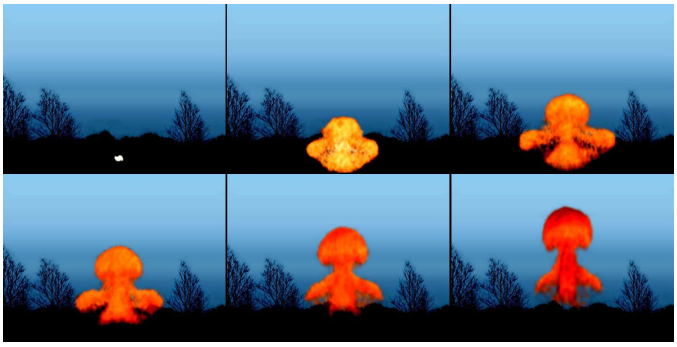

Figure 3.4: Six time steps from a particle based explosive flame [TOT+03]

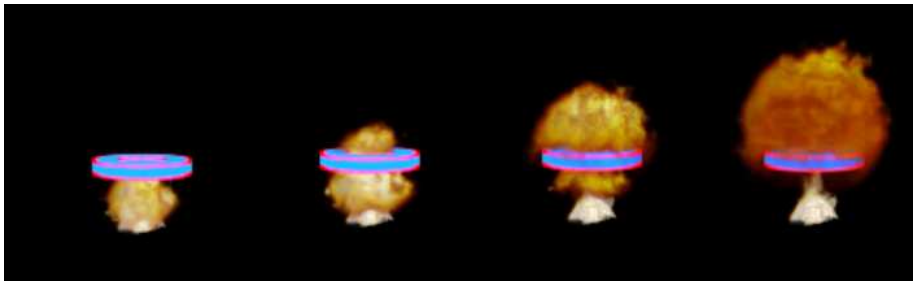

Figure 3.5: Torus shaped pressure template used to simulate an explosion [KW05]

3.1.3 Combustion

There exist several approaches that simulate phenomena that include combustion. However, many of these approaches do not model the combustion explicitly. [RNGF03] models detonations and assume that the combustion has completed at the beginning of the simulation. [KW05] simply uses colored dye that is carried along the fluid velocity to imitate the appearance of burning gas.

An explicit model is used by [MK02] to simulates fire. The computation is represented by a grid structure, and the combustion process is simulated in each of the grid cells. If the temperature in a cell is high enough, both the fuel gas, exhaust gas, and temperature are updated based on a combustion parameter. This combustion parameter is computed based on the amount of fuel and oxygen inside the grid cell. The increased temperature will diffuse to the neighbouring cells causing the combustion process to begin in these cells as well. The combustion can be adjusted by varying several parameters. These parameters include the burning rate, which is the percentage of the

fuel gas that can be burned in a second, the output heat from the reaction, and the stoichiometric mixture. The stoichiometric controls the amount of oxygen that is required for the combustion of one unit of fuel. If the present amount of air is insufficient, it will limit the combustion. [GRS06] used the combustion model of [MK02], but without the oxygen requirement. Thus, they assume that there will always be enough oxygen to sustain the combustion process.

[FOA03] uses a similar, explicit model to simulate the combustion of small fuel particles. The particles ignite when their temperatures rise above a certain threshold. Once ignited, they burn at a constant rate and are removed from the simulation when their mass reaches zero. The burning fuel particles introduce heat and soot at rates proportional to the burning rate. Heat is added directly to the underlying fluid simulation, while soot is accumulated in a variable associated with the fuel particle. When enough soot is accumulated a soot particle is introduced to the simulation, and the soot variable is reset. As [GRS06], [FOA03] also assumes that there is always enough oxygen to sustain the combustion process.

3.1.4 Volumetric extrusion

A full 3D fluid simulation has high computational and memory requirements, and these requirements often put limitations on the possible simulation grid sizes that can be used in such a simulation. Often, smaller grid sizes than desired have to be used. A way around this problem is by the use of volumetric extrusion. [RNGF03] proposed a way to derive a 3D velocity field by combining two or more 2D velocity fields. The 2D fields were treated as slices in the 3D space, and the values in between the 2D fields were calculated using interpolation. The 2D fields were obtained from a 2D simulation, and thus, both the simulation time and the memory requirements were drastically reduced. Using this technique, highly detailed large scale phenomena, such as nuclear explosions, could be modeled using only low to medium amounts of memory.

Volumetric extrusion was also used by [KW05] when simulating volumetric effects such as smoke, fire, and explosions. They placed independently simulated 2D fields in a cylindrical form in the 3D domain, and used cylindrical interpolation to retrieve the values from the 3D domain. Using this technique in combination a fluid simulation all performed on the GPU, they were able to perform semi-3D fluid simulations in real-time.

3.1.5 Vorticity and turbulence

As already mentioned, [Sta99]'s method has a few drawbacks, one of which is the introduction of numerical dissipation. A way to make up for some of the numerical dissipation in Stam's simulation was introduced by [FSJ01]. They used a technique called vorticity confinement earlier introduced by [SU94] to locate and enhance existing vorticity. This way, the loss of vorticity due to numerical dissipation were less noticeable. The added vorticity gave great improvements compared to using Stam's fluid solver alone. Figure 3.6 shows an example from the simulation of rising smoke. Here, the clearly defined vortices are preserved to a large degree by the use of vorticity confinement.

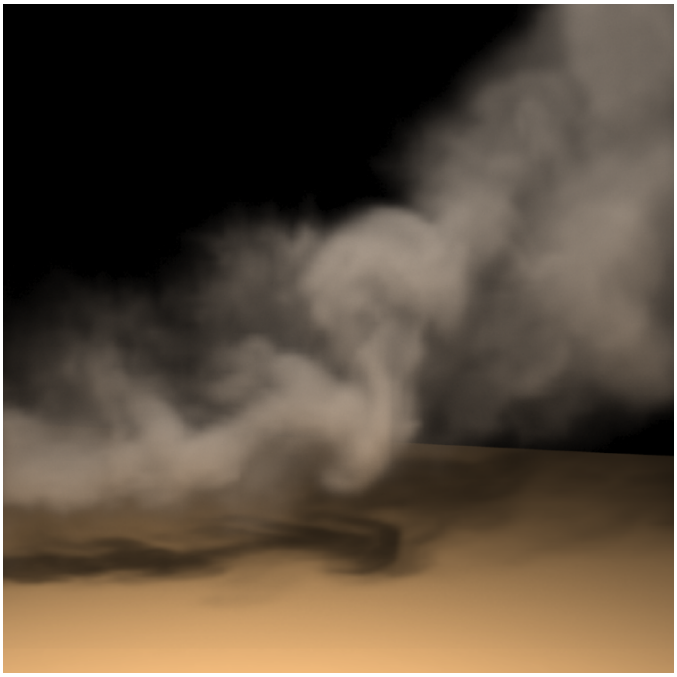

Figure 3.6: Simulation of smoke using Stam's method combined with vorticity confinement [FSJ01]

Another technique that can be used to reduce the numerical dissipation is Back and Forth Error Compensation and Correction (BFECC). BFECC was introduced by [DL03] as a level set computation method, and was applied by [KLLR05] to the velocity and density advection steps in Stam's method. The idea behind the technique is simple. If the advection operator used in Stam's method was correct, advecting forward and then backward would result in the same value as the original one. However, the advection operator introduces an error. This error is calculated and compensated for in a final forward advection step. The technique is easy to implement on top of the

semi-Lagrangian integration already used, and produces impressive results, as can be seen in figure 3.7.

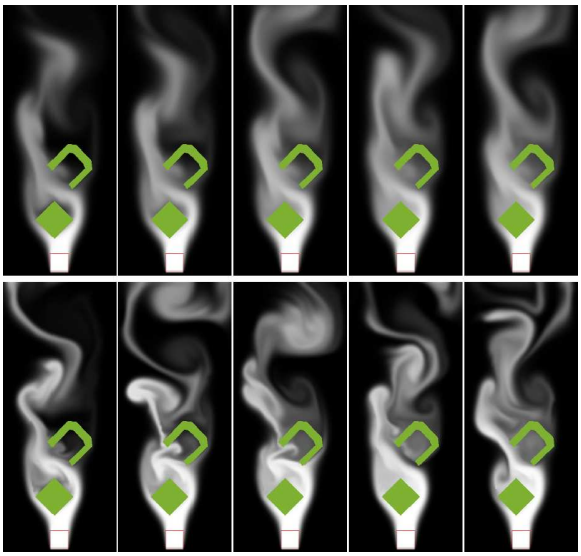

Figure 3.7: Comparison of smoke simulation with (bottom) and without (top) BFECC [DL03]

Even though vorticity confinement can enhance existing vorticity, it is not able to introduce new vorticity to the fluid. [SRF05] actually stated that vorticity confinement alone can be insufficient for highly turbulent effects, such as rough water or explosions. Instead, they proposed a hybrid method between the standard semi-laplacian methods, such as the one by Stam, and the Lagrangian vortex particle methods. Their method includes an Eulerian grid of velocities in addition to a set of vortex particles whose motion were affected by the velocity field. Each vortex particle has a certain vorticity that can be used to affect the velocity field in several ways, and thus introducing additional turbulence to the fluid. Figure 3.8 shows the evolution of smoke enhanced with vortex particles.

Another way to introduce additional turbulence is by the use of a Kolmogorov spectrum. As mentioned earlier, instead of performing a full 3D simulation, [RNGF03] performed a few 2D simulations and interpolated between the resulting 2D fields to approximate a simulation in the 3D space. The velocities from the 2D fields were combined with an additional velocity field derived from a Kolmogorov spectrum to introduce additional low level detail. Two or more such Kolmogorov velocity fields were used, each assigned to different points in time and interpolated between. This way, the turbulent field into which the particle motion were transitioned into var-

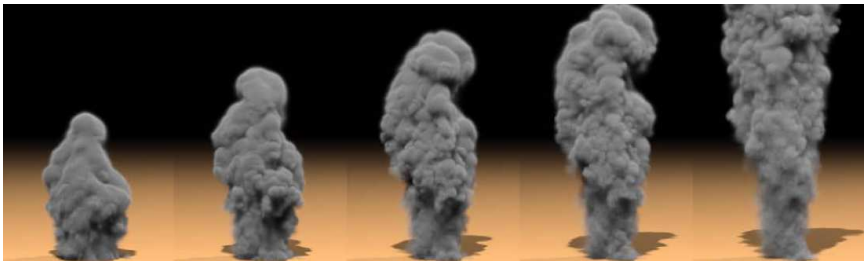

Figure 3.8: Simulation of smoke using the Vortex Particle method on top of the Stable Fluid solver.

ied over time. An example showing an explosion where such a Kolmogorov velocity field is used can be found in figure 3.3.

3.1.6 Arbitrary boundary conditions

Most real-time simulation that include fluids only consider the boundaries at the edges of the simulation domain. This reduces the complexity of the code that needs to be executed per cell in the simulation domain. However, allowing for arbitrary boundary conditions opens for numerous interesting effects.

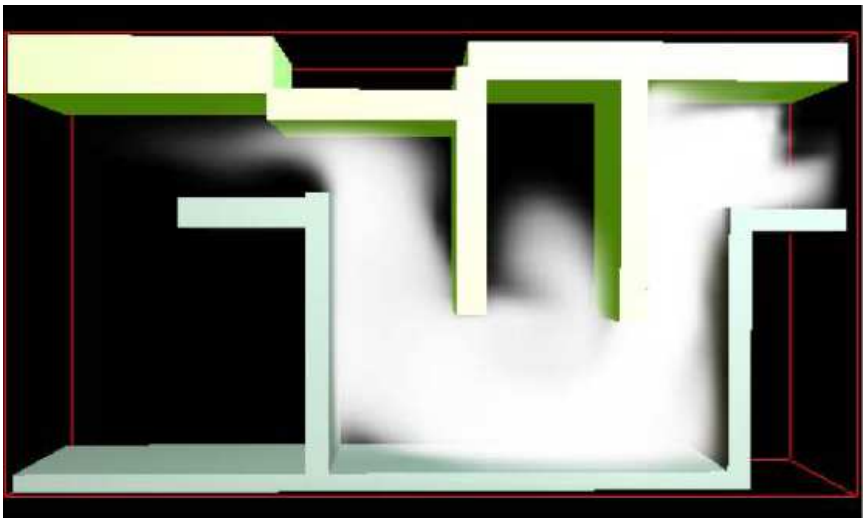

Figure 3.9: 3D smoke simulated using internal boundaries [LLW04].

[WLL04] combine the Stable Fluids solver with a more flexible way of treating arbitrary boundary conditions. This is used to simulate the motion of a

2D fluid that flows around obstacles. In [LLW04], they extend their work to 3D. The GPUs they had at their disposal did not allow branch operation, a fact their solution is influenced by. To avoid branch operations[1] in pixel shaders they encoded offset and modification data in textures and created pixel shader code that used this data to perform the complete algorithm on the GPU. Figure 3.9 shows an example from their results.

Figure 3.10: 3D smoke simulated using moving internal boundaries [LLW04].

A more recent approach can be found in a sample from NVidia [NVi07]. As current GPUs support branch operations natively, a far more intuitive solution can be implemented. They use textures to represent whether or not a cell is occupied by an obstacle and at what speed the obstacle is moving. This information is sampled by a simple texture fetch and branching is used to decide if the cell should be handled as a boundary cell or not. Figure 3.10 shows an example of their results.

[1] In this section, branch operations refer to if-tests.

3.2 Visualization of explosions

The techniques used to visualize explosions basically use the data that results from the explosion simulation. These data include the velocities, the exhaust gas, and the temperature. Simulation of other volumetric effects, such as fire and smoke, often use the same kind of data, and thus the various visualization techniques should be able to visualize a range of different volumetric effects. For that reason, this overview will not discuss related work to the visualization of explosions alone, but also include techniques used to visualize other related volumetric effects as well.

There are mainly two approaches that are used in real-time visualization of volumetric effects; particle systems, and volume rendering. Other approaches that may produce more visually pleasing results exist, but they are often too computationally expensive to be used for real-time visualization. For that reason, they are often referred to as offline approaches.

In the remainder of this section, the mentioned approaches will be presented. Techniques that use particle systems will be presented first, followed by an overview of techniques based on volume rendering.

3.2.1 Particle systems

Particle systems offer a relatively easy way to visualize volumetric effects. They have been used for several decades, and one of the earliest work can be found in [Ree83]. In more recent years, particle systems have been used both for real-time visualization, such as in [WLMK02], [KSW04], [Lat04], and [KW05], and for offline visualization, as in [FOA03].

3.2.1.1 Simulation of particle systems

The particles in a particle system can either be specific to and injected in the visualization step, as in [KW05], or be an active part of the simulation itself, as in [FOA03]. When the particles are visualization specific and separate from the simulation, the particles are often advected by a velocity field, which is typically calculated in the simulation step prior to the visualization step. The velocities can be a discretized 3D velocity field, as in [WLMK02], or one or more discretized 2D velocity fields, as in [KW05]. Since the velocity fields are discretized, the velocity vectors used to move the particles have to be calculated by interpolating the velocity fields based on the position of the particle relative to the velocity field. When a 3D velocity field is used, the velocity vectors can be calculated using tri-linear interpolation as

in [WLMK02], whereas when 2D velocity fields are used, the velocity vectors can be calculated using volumetric extrusion as in [KW05].

3.2.1.2 Visualization of particle systems

When the particles are rendered, they can be rendered as points, as in [FOA03], or as larger point sprites, as in [WLMK02] and [KW05]. When using larger point sprites, visually pleasing results can be achieved by using less particles, resulting in better performance. Large point sprites will however reduce some of the fine-scale detail that may be more visible when small points are used. A way to introduce additional fine-scale detail even when larger point sprites are use is to use render the sprites using textures splats, as in [WLMK02].

The colors of the particles can vary based on various quantities, such as the density or the temperature. This technique is among others used by [WLMK02]. First, they decide whether a particle is to be rendered as fire or as smoke based on the temperature at the particle's position. If the temperature is above a certain threshold, the particle is rendered as fire, and the color is calculated using a black-body radiation model based on the given temperature. Otherwise, the particle is rendered as smoke using a different color range.

When the particles are rendered on top of each other, the colors are blended to create smooth transitions between the particles. However, rendering the particles in an arbitrary order may cause visual artifacts. These visual artifacts can be removed if the particles are sorted back-to-front before they are rendered. This technique is used both by [WLMK02] and [KSW04].

3.2.1.3 Implementing particle systems on the GPU

With the introduction of modern graphics hardware, a particle system can be implemented in parts or fully on the GPU, as done both by [KSW04] and [Lat04]. Here, the positions and velocities of the particles are stored in separate textures, where the various color channels are used to store the x-, y-, and z-coordinates. Since a texture can not be used both for input and output, a pair of textures are used for each of the quantities, where one of the textures is used for input and the other texture is used for output. The role of the textures are switched every frame, so that the output texture in one frame is used as input in the next frame, and vica versa.

The movement of the particles is performed in two steps. First, the velocities are updated based on some local or global forces, before the positions are

updated based on the velocities. In [Lat04], only parts of the simulation is performed on the GPU, as the particle spawning and removing is performed on the CPU.

The particles can be rendered as points, triangles, or quads, as in [Lat04]. Here, a static vertex buffer is drawn, and the vertex positions are updated based on the particles' positions. These position are located in a texture, and are fetched using vertex-texture-fetch functionality. Also, to avoid visual artifacts, both [KSW04] and [Lat04] sort the particles the particles before they are rendered.

3.2.2 Volume rendering

One way to do volume rendering was shown by [IMDN05]. In their work, they place a texture at the center of each voxel of the voxelized data from the simulation. These textures are then rendered using the GPU. The color of the textures are calculated using a black-body radiation model based on the temperature.

Another similar technique is to replace every voxel with a semi-transparent polygon. The density of the exhaust gas and the fuel gas decide the level of transparency in the voxel. Also, the amount of fuel combustion at the voxel decides the intensity of the red fire color, whereas the amount of exhaust gas at the voxel decides the intensity of the smoke.

Ray marching is another technique used to do volume rendering, and is very suitable for use on the GPU. This technique is used by [KW03]. They cast rays through the volume, and map one pixel shader program to each ray. For each ray, they perform a number of steps through the volume. At each step, a scalar value is sampled from the volume, and this value is either used directly or in combination with a color lookup table to find another color value that is blended with the already accumulated color value. By casting rays uniformly through the entire volume and calculating the color values in this matter, the entire volume will be rendered.

3.3 Approach comparison

Figure 3.11 shows a table that compares different methods for animation of explosions in light of different aspects. An approach receives a mark if the it fulfills the given aspect. Real-time performance is defined as when the animation runs at at least 30 frames per second. As can be seen, only [KW05] performs in real-time.

Approach			Simulation			Rendering	
	Real-Time	Volumetric extrusion	Stable Fluids	Combustion	Thermal expansion	Particle based rendering	Volume rendering
FOA03			X	X	X	X	
RNGF03		X	X				X
TOT+03							X
KW05	X	X	X			X	

Figure 3.11: A comparison of different approaches for animation of explosions.

3.4 Summary

Approaches for real-time and offline simulation of explosions, as well as related phenomena, have been presented. Most approaches use fluid dynamics to guide the simulation, and Stam's Stable Fluids solver is probably the most popular method for real-time fluids. The Stable Fluids method suffers from numerical dissipation, which can be remedied by the vorticity confinement method. Fluid simulation are often affected by buoyancy and gravity to include the relevant external forces. Fire and explosion simulations may also include explicit combustion models to enhance the realism of the result.

Additional turbulence may be added by either the vortex particle method or by using Kolmogorov turbulence. To reduce the computational load of 3D simulations, 2D slice simulations may be performed instead. A 3D volume may then be obtained from these using volumetric extrusion.

Relevant methods for visualization of volumetric data may be roughly categorized into either particle based methods or volume rendering. Particle based methods generally use a set of discrete particles that move in the simulation domain. They can be rendered directly as points, but are often rendered as textured quads or as spherical particles by more complex methods. Volume rendering, on the other hand, seeks to visualize the data more directly by sampling several points in the data per pixel that is rendered.

Chapter 4
Simulating Explosions

This chapter gives the details of the suggested method for simulating real-time explosions, whereas the next chapter explains how the simulation is visualized. The simulation method is similar to the fire simulation from [GRS06], in which it combines a combustion model with the Stable Fluids solver [Sta99] and vorticity confinement [FSJ01]. To model the thermal expansion that occurs during combustion, the fluid solver is modified as suggested by [FOA03]. Additional turbulence is introduced by the vortex particle method [SRF05]. In order to restrict the computational requirements when simulating rotational, symmetric explosions, volumetric extrusion [RNGF03] is utilized[1].

The chapter is organized as follows: First, an overview of the simulation is given. Next, the details of the simulation fields and the various equations governing them are presented. Then, boundary conditions are discussed, followed by a presentation of the complete simulation algorithm. Finally, methods for sampling from the simulation results are explained.

4.1 Overview

The explosion simulation models the combustion of fuel that produces exhaust gas and temperature changes in a computational domain. The fuel, exhaust gas, and temperature values are scalars that reside in separate fields, but they are collectively referred to as the "density fields", or simply "densities". Exhaust gas and temperature affect a velocity field by gravitational and buoyant forces, and the velocity field is in turn used to move the contents of the density fields. The velocity field contains vectors that describe the direction and speed of the fluid, the air in which the explosion occurs. The various fields are discretized into grid or voxel cells representing a simulation domain in either 2D or 3D, respectively. The fields are governed by a set of differential equations, which are solved on a cell by cell basis.

[1] Volumetric extrusion is also used by [KW05, GRS06]

4.1.1 Computational domain

When the simulation is performed in 2D, the computation is represented by a grid structure, as shown by figure 4.1. Each cell, shown as squares, will contain corresponding density and velocity field values defined in cell centers. Such a representation is known as a collocated grid, as opposed to staggered grids. When using a staggered grid representation, vertical velocity is defined at horizontal cell borders, horizontal velocity is defined at vertical cell borders, while density values are still defined at cell centers. The implementation when using collocated grids are more straighforward [Sta99].

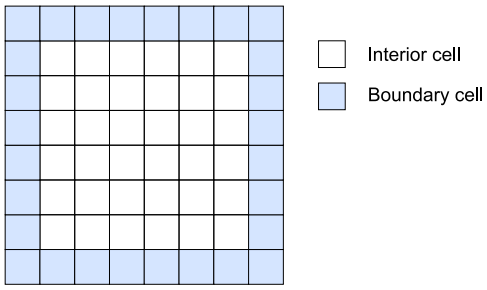

Figure 4.1: A 2D computational domain [SR06].

When the simulation is performed in 3D, a voxel representation is used. A cross-section of such a 3D domain is shown in figure 4.2. Similarly to the 2D representation, voxels in the 3D representation have corresponding density and velocity field values defined in voxel centers. 2D grid cells and 3D voxels are collectively referred to as *grid cells*, or simply *cells* from now on.

When solving differential equations, one often needs to look at neighboring cells. Cells at the boundary of the simulation domain does not have neighboring cells in every direction, thus certain considerations need to be made for these cells. Figures 4.1 and 4.2 make distinction between two kinds of cells; interior cells and boundary cells. All the equations in the simulation can be calculated for the interior cell, while the boundary cells are subject to *boundary conditions*. The boundary conditions vary based on the kind of numerical operation that is performed and are explained in detail in section 4.3.

4.1.2 Simulation overview

Figure 4.3 shows a conceptual overview of the simulation method. The simulation state that is updated and kept from one time step to the next is shown

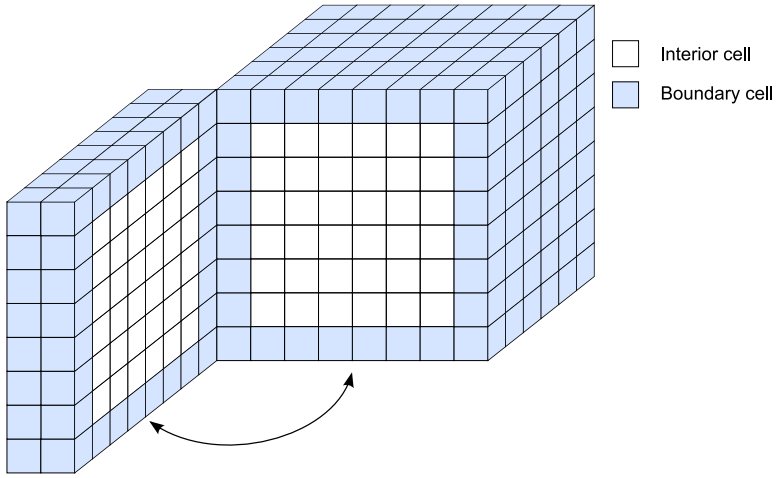

Figure 4.2: A cross-section of a 3D computational domain [SR06].

in the frame marked *simulation state*. The state consists of fields, shown as boxes, and a table of vortex particles. The three density fields, fuel, exhaust, and temperature, are situated to show their coherence. The main processes, shown as circles, are categorized into either *explosion specific* or *fluid solver* processes. The simulation also contains fields that hold intermediate results. These are not shown in the figure, but are rather explained subsequently in appropriate sections. An arrow from a state object to a process expresses that the state object data is read by the process, while an arrow going from a process to a state object means that the state is updated as a result of the process.

The actual order of process execution is dictated by both the fluid solver itself and the fact that the output of some processes is input data to others. Several valid orders exist. In fact, some may even be performed in parallel. Figure 4.4 shows how the processes from figure 4.3 relate to each other. The *advection* process in figure 4.3 occurs twice in the actual method and is therefore shown more detailed by the *velocity advection* and *density advection* processes in figure 4.4. An arrow going from a process to another means that the first process needs to be completed in order for the second to start. The vertical lines illustrate how processes may be grouped for parallel execution. Also, figure 4.4 shows two respective groups of four and two processes. This is to illustrate the processes' logical coherence, and the fact that they all produce results that another process depends on.

The order in which the processes are presented below is chosen to ease the explanation:

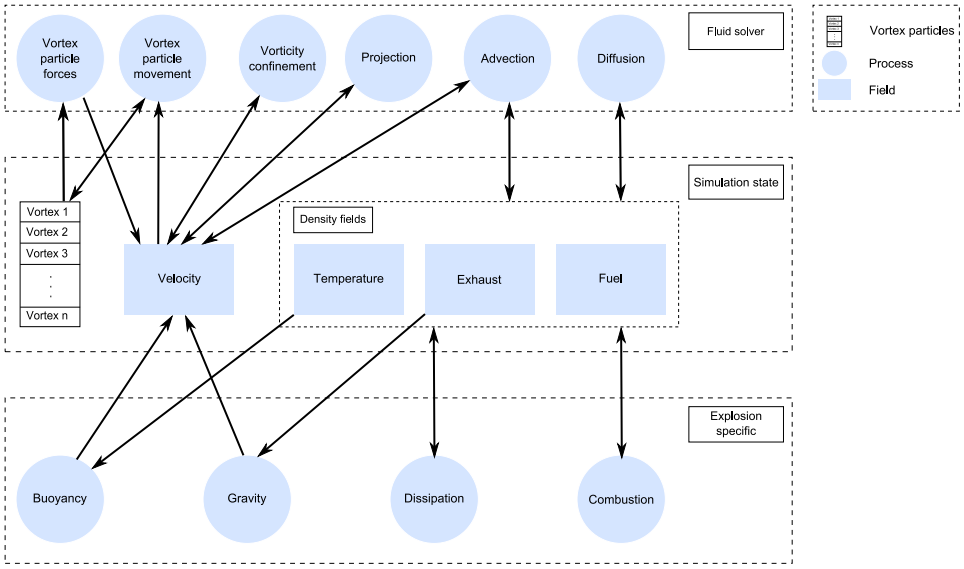

Figure 4.3: Conceptual simulation overview

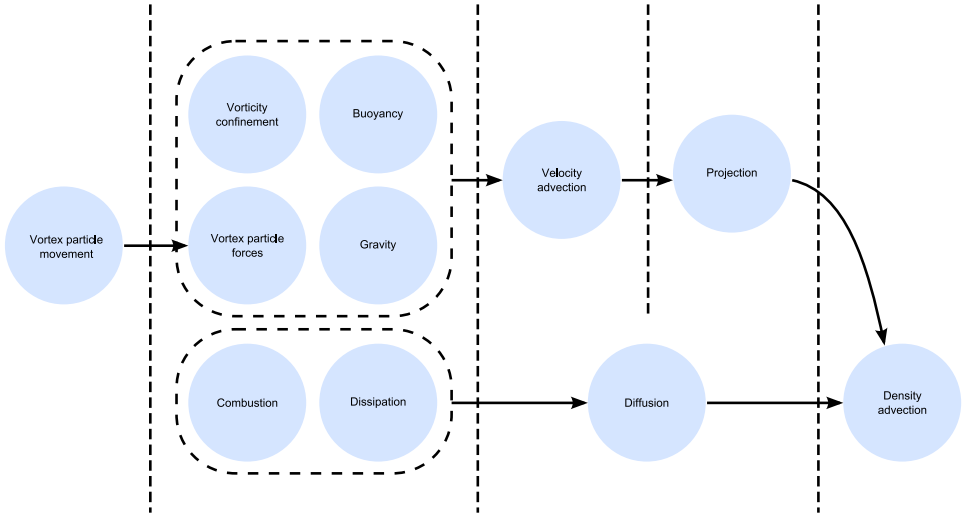

Figure 4.4: The processes and their relationship to each other.

First, the *Vortex particle movement* process moves a set of vortex particles based on the underlying velocity field. The vortex particles themselves are used later to create rotational motion in the simulation. The method is explained in detail in section 4.2.3.3.

Secondly, forces that affect the velocity of the fluid are calculated and added to the velocity:

- *Gravity* pulls dense exhaust gas towards the ground.
- *Buoyancy* due to high temperature causes the fluid to rise.
- *Vorticity confinement* locates rotational motion in the fluid and enhances it.
- *Vortex particle forces* create additional rotational motion.

Third, changes to the density field are performed:

- *Combustion* models burning fuel, creating exhaust gas and heat.
- *Dissipation* models dissipation of fuel, exhaust gas, and temperature field, making the explosion calm down after a while.

Fourth, the fluid solver evolves the velocity field:

- *Advection* moves the velocity field based on the velocity itself, often referred to as self-advection.
- *Projection* ensures that the velocity field has the desired divergence, which involves an intermediate field that models the thermal expansion.[2]

Finally, the fluid solver moves the density and incorporate the effect diffusion:

- *Diffusion* models the fuel, exhaust gas, and temperature's tendency to spread.
- *Advection* moves the contents of the density fields based on the velocity.

[2]Note that to keep figure 4.3 as simple as possible, the thermal expansion field is not shown.

4.2 Simulation details

This section provides the details behind the processes just explained. Note that several of the aspects covered in following sections expect knowledge of the Stable Fluids solver that were presented in 2.2.3.

The section is organized as follows. First, the equations governing the velocity and density fields are presented. Next, the various forces that affect the fluid motion is explained. Then, the combustion model is presented, followed by an explanation of thermal expansion is calculated.

4.2.1 Velocity fields

The fluid velocity field is governed by the Navier Stokes equations for incompressible inviscous flow:

$$\frac{\delta \mathbf{u}}{\delta t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f} \tag{4.1}$$

Equation 4.1 states that the change of the velocity $\mathbf{u}$ consists of the three terms on the right hand side of the equation. The first term represents the advection that move the fluid velocity along by itself. The second term models the effect of pressure, causing fluid to flow from areas of high pressure to areas of low pressure. The third term provides a way to include external forces to the fluid. The force term is actually a combination of forces, whose details are covered in section 4.2.3.

When assuming the fluid is incompressible, mass conservation laws require that the velocity divergence is zero. However, during combustion the temperature of the fluid will increase and cause thermal expansion. This effect is approximated as in [FOA03] and we allow for:

$$\nabla \cdot \mathbf{u} = \phi \tag{4.2}$$

where ϕ is proportional to temperature changes.[3] Thus, divergence will still be zero in areas where the temperature is constant, but the divergence in areas where the temperature rises will be positive. Combustion will cause the temperature to rise quickly and lead to high velocities away from the cells involved, which imitate the violent motion of explosions.

[3] Note that the proportionality constant must be positive. If not, the thermal expansion model would result in an *implosion*, not an explosion.

The pressure term of equation 4.1 is not modeled explicitly, but accounted for by the projection process of the fluid solver to ensure that equation 4.2 holds. Because the usual $\nabla \cdot \mathbf{u} = 0$ condition of incompressible fluid has been replaced with equation 4.2, certain changes to the projection step need be taken. Instead of solving

$$\nabla \cdot \mathbf{w} = \nabla^2 q \tag{4.3}$$

a slightly modified poisson equation is solved instead:

$$\nabla \cdot \mathbf{w} - \phi = \nabla^2 q \tag{4.4}$$

In practice, the values of the symbols on the left side of equation 4.4 can be calculated before letting the fluid solver find q. Thus, the extension of the fluid solver to include thermal expansion does not involve substantial additional computations.

The density ρ of equation 4.1 is assumed to be constant. One could include varying density in the simulation to model expansion as performed in [KLLR05], at the expense of a slightly more complex poisson equation to solve. Since the method from [FOA03] produces the desired results, the method from [KLLR05] has not been an area of focus.

Like several others [FOA03, RNGF03, GRS06] the viscosity term of the Navier Stokes equations is omitted, as it will have negligible effect on the final result . Viscosity models the flow resistance of a fluid and is important when modeling fluids like syrup or oil, but can be ignored when dealing with gases.

4.2.2 Density fields

As [Sta99] proposed, the velocity field can be used to move densities that reside in the fluid. He also proposed diffusing these densities to model fluid's tendency to spread. In this model, fuel, exhaust gas, and temperature values represent these densities. Similarly to [Sta99, GRS06], density evolution over time is governed by the following equation:

$$\frac{\delta d}{\delta t} = -(\mathbf{u} \cdot \nabla)d + \kappa_d \nabla^2 d - \alpha_d d + C_d \tag{4.5}$$

where d represent either the fuel g, the exhaust gas e, or the temperature T. Thus, equation 4.5 yields three differential equations; one for each density

quantity.

The first term of equation 4.5 is the advection term, where the densities are moved along the velocity field $\mathbf{u}$. The second term is the diffusion term, where each density is diffused using their respective diffusion constants κ_g, κ_e, or κ_T. The third term models the dissipation of the densities, enabling the densities to slowly fade away at their respective dissipation rates α_g, α_e, or α_T. Intuitively, exhaust gas may drift for a while, but will eventually disperse. The temperature is dissipated to model the radiance and conduction of hot gas. The fuel dissipation constant is kept low, or even ignored because the fuel will combust during the first few moments of simulation. The final term is the combustion term, which include the results provided by the combustion model (explained in section 4.2.4).

4.2.3 Forces

The differential equation for the evolution of velocity is general in which it may be applied to a wide variety of phenomena. To simulate the motion of an individual effect, like explosions, one has to apply external forces to equation 4.1 that describe the phenomenon. The external force term, $\mathbf{f}$, of the equation is a combination of several others as shown in the following equation:

$$\mathbf{f} = \mathbf{f}_{gravity} + \mathbf{f}_{buoyancy} + \mathbf{f}_{vorticity} + \mathbf{f}_{particles} \tag{4.6}$$

where $\mathbf{f}_{gravity}$ and $\mathbf{f}_{buoyancy}$ are forces that pull the fluid towards the ground and make it rise, respectively. $\mathbf{f}_{vorticity}$ is a vorticity confinement force used to enhance existing rotational motion in the fluid, while $\mathbf{f}_{particles}$ is a force included to introduce additional turbulent motion to the fluid. The individual forces are explained in detail in the sections below.

4.2.3.1 Gravity and buoyancy

An important visual quality of explosions is their rising buoyant plume. This is modeled by including a thermal buoyancy force proportional to the temperature in the cells, as shown by the following equation:

$$\mathbf{f}_{buoyancy} = f_b T \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{4.7}$$

where T is the temperature of a cell and f_b is a positive constant used to control the strength of the buoyancy force.

The exhaust gases are assumed to be more dense than air, thus is pulled towards the ground by gravitational forces. Fuel gas, on the other hand, is assumed to be equally dense as the air and is ignored in gravity calculations. The following equation shows how the gravity force is calculated:

$$\mathbf{f}_{gravity} = f_g e \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{4.8}$$

where e is the exhaust gas level and f_b is a positive constant used to control the strength of the gravity force.

Buoyancy and gravity have previously been modeled similarly by several other methods, both for simulating explosions [FOA03, RNGF03] and related phenomena [FM97, MK02, IMDN05].

4.2.3.2 Vorticity confinement

When using the Stable Fluids solver, some of the small-scale rotational motion of the fluid is lost, an effect known as numerical dissipation, and results in a less turbulent fluid. To counterbalance this, the vorticity confinement method [FSJ01], also used by [RNGF03, FOA03, GRS06], is utilized to enhance existing rotational movement, or vortices, in the fluid. First, the existing vorticity ω of the velocity field $\mathbf{u}$ is found using the following equation:

$$\omega = \nabla \times \mathbf{u} \tag{4.9}$$

The vorticity ω is a vector whose magnitude describe the amount of rotational motion. Its direction describe the axis of which the fluid rotates around in a right-handed fashion. Next, a normalized vector $\mathbf{N}$ pointing toward areas of greater magnitude of vorticity is calculated:

$$\mathbf{N} = \frac{\nabla |\omega|}{|\nabla |\omega||} \tag{4.10}$$

To be able to enhance existing vorticity, a force term whose direction is tangential to the circular motion is needed. Such a vector can be found to be the cross-product of ω and $\mathbf{N}$:

$$\mathbf{f}_{vorticity} = \varepsilon h(\mathbf{N} \times \omega) \tag{4.11}$$

where h is the distance between adjacent cell centers and ϵ is a positive scalar used to control the strength of the vorticity confinement strength.

4.2.3.3 Vortex particles

Vorticity confinement does only enhance existing rotational motion in the fluid, thus it will not be able to *introduce* turbulence to the simulation. [SRF05] proposed a vortex particle method where particles add rotational motion to their surroundings, while they flow through the fluid, guided by the velocity field. The vortex particle method is utilized to introduce turbulence to the simulation domain through the force $\mathbf{f}_{particles}$ of equation 4.6.[4]

Since vorticity is defined as the curl of the velocity ($\omega = \nabla \times \mathbf{u}$), one can take the curl of equation 4.1 to yield a differential equation that describe how the vorticity of the velocity field evolves:

$$\frac{\delta\omega}{\delta t} = -(\mathbf{u} \cdot \nabla)\omega + (\omega \cdot \nabla)\mathbf{u} + \nabla \times \mathbf{f} \tag{4.12}$$

The changes of vorticity during a small time step δt is dictated by the terms of the right side of the equation. The curl of the advection term of equation 4.1 yields the two first terms; the first is the vorticity advection term, describing the movement of vortices in the fluid; the second is a vortex stretching term that describes changes to the vortex vector itself, in the form of both direction and magnitude. Note that the pressure gradient term of equation 4.1 vanishes, because the curl of a gradient field is to zero[5]. [SRF05] argue that the third term of equation 4.12 can be omitted because the forces are already applied to the fluid through equation 4.1.

Equation 4.12 is not solved numerically like equation 4.1. Instead, a number of vortex particles are defined with respective positions and vorticity vectors ω. They are passively advected by the fluid velocity at particle positions as shown by the following equation:

$$\mathbf{x}_p^{new} = \mathbf{x}_p + \mathbf{u}(\mathbf{x}_p)\delta t \tag{4.13}$$

[4]In contrast, turbulence can also be included to the visualization without affecting the simulation itself. This technique is also used, and is explained in chapter 5.

[5]This is a mathematical fact and can be shown simply by calculating $\nabla \times \nabla p$, where p is a scalar field.

This accounts for the advection term of equation 4.12. The vortex stretching term, on the other hand, has an effect on the particles' vorticity vector. The vorticity vectors are updated each simulation step using the following formula:

$$\omega^{new} = \omega^{old} + (\omega^{old} \cdot \nabla)\mathbf{u} \tag{4.14}$$

where finite difference derivation is used on the velocity field to evaluate the vortex stretching term.

[SRF05] also informs that the vortex particle magnitude could increase exponentially when the vortex stretching term has positive eigenvalues based on the fluid velocity gradient. To avoid this, the magnitude is clamped, only allowing it to decrease.

Note that when the simulation is performed in 2D it is assumed that only the z-component of ω is non-zero. Also, $\frac{\delta \mathbf{u}_x}{\delta z}$ and $\frac{\delta \mathbf{u}_y}{\delta z}$ evaluate to zero. Thus, the only non-zero value of the vortex stretching term is the z-component, which evaluates to ω itself. This leads to the exponential increase that [SRF05] mentioned, but as long the magnitude is clamped the complete vortex stretching term can be omitted.[6]

Each particle emits vorticity to its surrounding fluid. A distribution kernel $\xi(\mathbf{x} - \mathbf{x}_p)$ is used to define the amount of vorticity that is induced by a particle at $\mathbf{x}_p$ to a arbitrary position x. The vorticity induced is:

$$\tilde{\omega}_p(\mathbf{x}) = \xi(\mathbf{x} - \mathbf{x}_p)\omega_p \tag{4.15}$$

By choosing a kernel that is rotational symmetric and strictly decreasing a normalized vector $\mathbf{N}_p = \frac{\mathbf{x}-\mathbf{x}_p}{|\mathbf{x}-\mathbf{x}_p|}$, which correspond to the $\mathbf{N}$ of the vorticity confinement method, can be computed. Similarly, the force induced by a single vortex particle is:

$$\mathbf{f}_p = \mathbf{N}_p \times \tilde{\omega}_p \tag{4.16}$$

The contributions of forces from all the vortex particles are summed up to find the final vorticity force $\mathbf{f}_{particles}$ at each cell in the simulation domain.

[6] Note that we have only implemented the vortex particle method in 2D.

4.2.4 Combustion

A central aspect of effects like explosions and fire is combustion. The combustion model from [MK02], slightly modified by [GRS06], is used to simulate burning fuel, generating exhaust gas and thermal energy.

The combustion will only occur if the temperature T is above the lower ignition limit $T_{threshold}$, which correspond the the explosion physics described in section 2.1. The model assumes that there will always will be enough oxygen available to react with the fuel; an assumption not generally valid, but certainly valid for some types of explosions such as dust explosions [FOA03].

The model calculates a combustion parameter, which is used in the subsequent calculation:

$$C = \begin{cases} rbg & \text{if } T > T_{threshold} \\ 0 & \text{elsewhere} \end{cases}$$

where r is the burning rate used to describe the percentage of fuel that can be consumed in a second, b is a stoichiometric mixture constant that describe the amount of oxygen that reacts with one unit of fuel, and g is the amount of available fuel. Equations 4.17, 4.18, and 4.19 use C to describe the respective changes to fuel, exhaust gas, and temperature:

$$C_g = -\frac{C}{b} \tag{4.17}$$

$$C_e = C(1 + \frac{1}{b}) \tag{4.18}$$

$$C_T = T_0 C \tag{4.19}$$

The stoichiometric mixture constant b can be adjusted to alter the amount of exhaust gas that is generated by the combustion. Equation 4.19 uses a heat generation constant T_0. It is, in combination with b, responsible for the amount of heat generated by the combustion.

C_g, C_e, and C_T are used in their respective versions of equation 4.5.

4.2.5 Thermal expansion

An important aspect of modeling explosions is to include the thermal expansion that occurs during the combustion. A thermal expansion value, ϕ, is calculated for each cell based on the change of temperature that is caused by combustion and dissipation:

$$\phi = \tau(-\alpha_T T + C_T) \tag{4.20}$$

A positive constant, τ, controls the amount of expansion. The first term of the parenthesis, $\alpha_T T$, is the temperature dissipation, which was explained in section 4.2.2. The second term, C_T, is the temperature change caused by the combustion process. Equation 4.20 is applied to each cell of the computational domain, resulting in a scalar field that represents the amount of expansion. The field is subsequently used to affect the divergence of the cells, as explained in section 4.2.1.

As can be seen from equation 4.20, the expansion, ϕ, may be negative when the dissipation term is larger than the combustion term. This is physically correct because the volume of gases that cool down decreases, as the reverse process of expansion caused by heat.

4.3 Boundary conditions

In general, boundary conditions are necessary when dealing with differential equations. In particular, when simulating differential equations on a grid structure finite differentiation needs the values of a cell's neighbors. This poses a minor problem not only on the bounds of the simulation domain, but also if internal boundaries are included in the simulation. Different considerations need to be taken in the steps of the fluid solver based on which step that is performed and based on the kind of boundary condition that is chosen. The chosen model does not allow fluids to flow across boundaries at any time. This assumption may seem strange when simulating explosion in an outdoor environment, but is not a problem as long as the explosion is kept away from the bounds of the simulation domain.

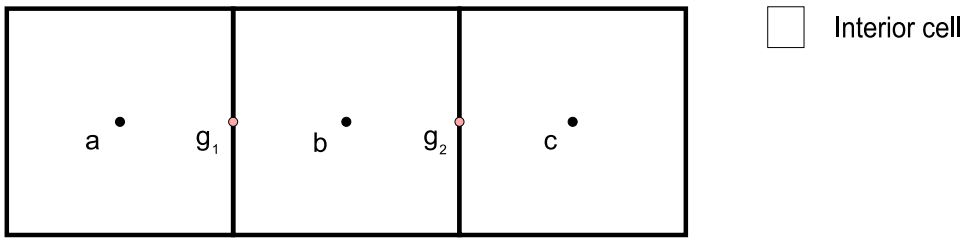

Figure 4.5: Three interior cells showing the positions of the relevant variables for finite differences derivate calculation.

Figure 4.5 shows three interior cells and the positions of the variables that are used to calculated the finite difference derivative in b. First the values

at $g_1 = \frac{a+b}{2}$ and $g_2 = \frac{b+c}{2}$ are found by averaging their neighbors. Then, the derivate d (in this example with respect to x) is found using the following equation:

$$d = \frac{(g_2 - g_1)}{h} = \frac{c - b}{2h} \tag{4.21}$$

where h is the size of the grid cells. Note that the figure and the equations only show the case at the right side of a simulation domain. The same principles apply to all bounds.

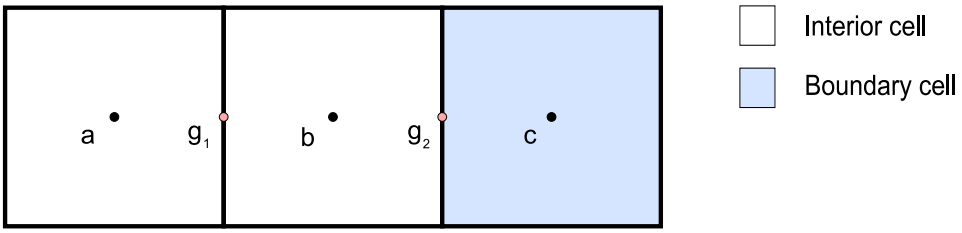

Figure 4.6: Two interior cells and a boundary cell showing the positions of the relevant variables for finite differences derivate calculation.

When one (or more) of the neighboring cells are boundary cells, as shown in figure 4.6, the values outside the boundary need to be decided (in the example figure this is variable c). In practice, the methods that used for choosing the values of the boundary cells fall into two categories; the first category contain divergence calculations, the other comprise all other operations.

4.3.1 Divergence operations

The projection step of the fluid solver calculates the divergence of the velocity field as a measure of velocity towards the cell minus velocity away from the cell. The calculation, $div(\mathbf{u}) = \nabla \cdot \mathbf{u}$, is performed by finite difference derivation as explained above. Since no fluid is allowed to flow across the bounds of the simulation domain, the perpendicular velocity at the borders must be zero. Exemplified by figure 4.7, this means that $\mathbf{u}_x(c) = -\mathbf{u}_x(b)$ must be chosen to ensure that $\mathbf{u}_x(g_2)$ evaluates to zero. This applies to the respective perpendicular components of all boundaries; Use the negative value of the cell itself in place of the cell outside the boundary.

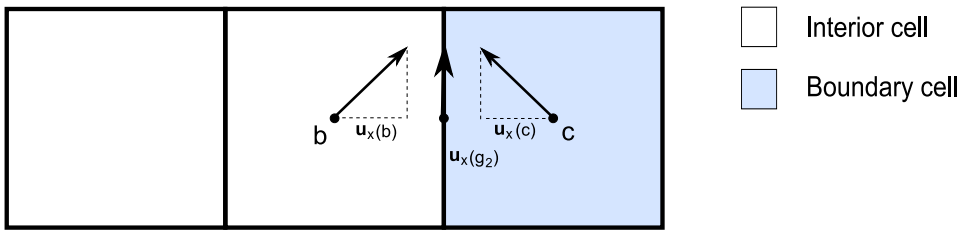

Figure 4.7: Shows how $\mathbf{u}_x(c)$ is chosen as $-\mathbf{u}_x(c)$.

4.3.2 Other operations

All other operations in the fluid solver that require boundary conditions are performed on scalar fields. The operations are; gradient operations, implicit diffusion operations, and poisson solver operations with Neumann boundary conditions. Common to all these is that they require the derivative at the boundaries to be zero. This can be ensured by letting $c = b$, again using figure 4.6 as an example.

4.4 Complete algorithm

The equation and methods that are presented in this chapter are coupled with the Stable Fluids solver to create the complete explosion simulation. The simulation is performed each time step and each of the equations are applied to each cell of the computational domain. The algorithm is as follows:

1. *Vortex particle movement* - Move the vortex particles using equation 4.13.
2. *Velocity force calculation* - Calculate the external forces and add them to the velocity field.
   (a) Buoyancy using equation 4.7.
   (b) Gravity using equation 4.8.
   (c) Vorticity confinement using equation 4.11.
   (d) Vortex particle force by summing the induced forces from all particles using equation 4.16.
3. *Density change calculation* - Calculate and add the changes to the density fields.

   (a) Combustion using equations 4.17, 4.18, and 4.19.
   (b) Dissipation using the density fields and the dissipation part of equation 4.5.

4. *Thermal expansion calculation* - Calculate a thermal expansion field using equation 4.20.

5. *Velocity fluid solver step* - Self-advect and project the velocity field.

   (a) Advection using the velocity field and the Stable Fluids solver to solve the advection part of equation 4.1.
   (b) Projection using the modified projection step of the Stable Fluids solver with the thermal expansion field from step 4.

6. *Density fluid solver step* - Advect and diffuse the fuel, exhaust gas, and temperature fields.

   (a) Advection using the Stable Fluids solver and the newly calculated velocity field from step 5 to solve the advection part of equation 4.5.
   (b) Diffusion using the density fields and the Stable Fluids solver to solve the diffusion part of equation 4.5.

4.5 Sampling from simulation results

As mentioned earlier in section 4.1.1, the computational domain is discretized into a grid of voxel cells, where values are defined at cell centers. Thus, when sampling from arbitrary positions, it is advantageous to combine the values of the closest cells. In 2D and 3D respectively, bilinear and trilinear interpolation are used to calculate smooth transitions between the cells. In addition, volumetric extrusion is used to implicitly define a 3D domain from a 2D domain.

4.5.1 2D and 3D sampling

To sample from an arbitrary position in a 2D simulation domain, bilinear interpolation is used. The four closest cells based on the sample position are used to produce a weighted average. Similarly, trilinear interpolation is used in 3D, where the values of the eight closest cells are used.

The use of interpolation results in an approximation of a continuous simulation domain. This is especially important during visualization to reduce visual artifacts caused by fact that the simulation is discretized.

4.5.2 Volumetric extrusion

The computational expense of performing a full 3D simulation is quite high. Therefore, the volumetric extrusion method from [RNGF03] is utilized to enable one or more 2D simulations to implicitly define a 3D domain. The volumetric extrusion itself is rather computationally inexpensive. Thus, performing a few 2D simulations instead of a full 3D simulation frees computational power that can be used to increase the detail of the simulation domain. Figure 4.8 shows an example of how two 2D slices can be used to define a 3D domain that is rotationally symmetric around the vertical axis.

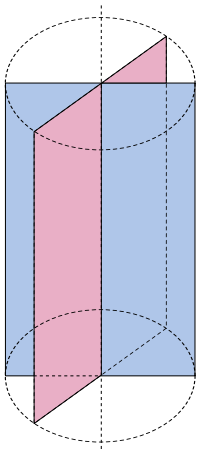

Figure 4.8: Two slices define a 3D volume through volumetric extrusion.

The values at points in the extruded 3D domain are computed through two interpolation steps; cylindrical interpolation is used between the two closest slices, while bilinear interpolation is used within the slices. Equation 4.22 and figure 4.9 show how to compute the sample value $S(\mathbf{x})$ at a position $\mathbf{x}$. The figure illustrates two slices that are oriented perpendicular to each other, as seen along the vertical axis. In the equation, α is the angle between the two slices, and $\mathbf{A}$ and $\mathbf{B}$ are corresponding positions in the two slices. $S(\mathbf{A})$ and $S(\mathbf{B})$ are computed using ordinary bilinear interpolation within the slices, while the cylindrical interpolation is defined by the interpolation factor $\frac{\alpha \cdot numslices}{\pi}$. The interpolation factor range from 0 to 1, thus a large angle α results in the dominance of $S(\mathbf{A})$ in $S(\mathbf{P})$, and vice versa.

$$S(\mathbf{P}) = \frac{\alpha \cdot numslices}{\pi} S(\mathbf{A}) + (1 - \frac{\alpha \cdot numslices}{\pi}) S(\mathbf{B}) \tag{4.22}$$

When constructing a 3D scalar field based on volumetric extrusion, the method just described can be used directly. However, when constructing

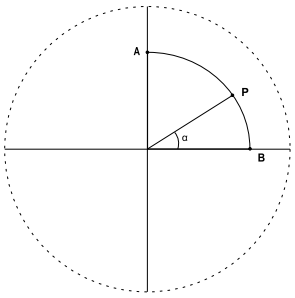

Figure 4.9: Top view of cylindrical interpolation (xz-plane).

a 3D velocity field based on 2D velocity slices, certain considerations must be made. The y-component can be sampled as described, but this is not the case for the xz-components. When using cylindrical interpolation, the x-component does not longer describe velocity in the x-direction, but rather the *magnitude* of velocity away from (or towards) the rotational axis. Also, the sign of x must be adjusted so it describes the velocity with respect to the rotational axis, which is where the slices intersect.

Let $\tilde{x}_{\mathbf{p}}$ be the scalar value of the x-component at $\mathbf{p}$ after both adjusting the sign and performing interpolation as described by equation 4.22. The *direction* of the the xz-component is then determined by $\mathbf{P}_{xz}/|\mathbf{P}_{xz}|$. The final velocity vector is found in the following equation:

$$\mathbf{u}_{\mathbf{p}} = \tilde{x}_{\mathbf{p}} \frac{\mathbf{P}_{xz}}{|\mathbf{P}_{xz}|} + \mathbf{y}_{\mathbf{p}} \tag{4.23}$$

where $\mathbf{y}_{\mathbf{p}}$ is the y-component in vector form, sampled as explained above.

4.6 Summary

A method for simulation of explosions has been described. It models the evolution of fuel, exhaust gas, and temperature field, as well as an underlying velocity field. The fields are governed by a set of differential equations, which are solved using Stam's Stable Fluids solver. Vorticity confinement is used to enhance rotational motion in the fluid, while the vortex particle method is utilized to introduce turbulence into the simulation.

A combustion model accounts for the burning of fuel, increasing the exhaust gas levels and temperature. To account for the thermal expansion that occurs during combustion, the Stable Fluids solver is modified to allow non-

zero divergence. Dense exhaust gas and temperature results in gravitational and buoyant forces respectively, which are used to affect the velocity field. In turn, the velocity field is used to move the contents of the fuel, exhaust gas, and temperature fields.

The simulation can either be performed in full 3D or by using volumetric extrusion to define a 3D domain through one or more 2D simulations.

Chapter 5
Visualizing Explosions

Several method for visualizing volumetric effects exists. Based on the real-time requirement, two main choices emerge: particle rendering and volume rendering. Particles can be rendered using texture splats to introduce small-scale detail. These details are difficult to obtain using volume rendering, especially without increasing the size of the simulation domain. Furthermore, the use of particle systems enables turbulence modeling to be utilized in the visualization step to reduce symmetry. This option is not easily accessible when using volume rendering. Finally, work performed by [SR06] on visualization of fire concluded that volume rendering currently fall short on both performance and visual quality, when compared to particle systems. All of the above led us to the decision of using particles to visualize the explosions.

The rest of the chapter begins with a brief overview of the visualization algorithm. Then, the details on how the particles are moved and how they are rendered are given. Finally, a complete step-by-step algorithm is given to summarize the visualization method.

5.1 Overview

The simulation results are visualized using two separate particle systems; one for fire and one for smoke. The particle systems consist of a fixed number of particles that move through the simulation domain guided by the fluid velocity from the simulation step and an independant turbulence model. All particles follow the same rules for movement, but we have included an additional level of control, which we call the movement factor.

Mainly, fire particles are visible when the temperature of exhaust gas exceeds a certain threshold, while smoke is visible first when the exhaust gas cools down. However, the threshold used in the actual visibility calculations is not strict, allowing a smooth transition from fire to smoke. The fire particles are rendered first, followed by smoke being blended on top.

The visualization algorithm is outlined in figure 5.1. Initially, particles are

created at individual positions. Then, following each simulation step, four visualization processes are performed. First, the particles are moved (process 1). Note that this step uses data from both the velocity field from the simulation step, as well as turbulence velocity from a turbulence vector field. Second, the new particle positions are saved, replacing the old particle positions (process 2). Finally, the fire particles are drawn to the screen, before the smoke particles are blended on top (processes 3 and 4). Both rendering processes use the density field from the simulation step to calculate particle colors and visibility. The particles are rendered as screen-aligned quads, textured with an animated texture to create a more dynamic appearance.

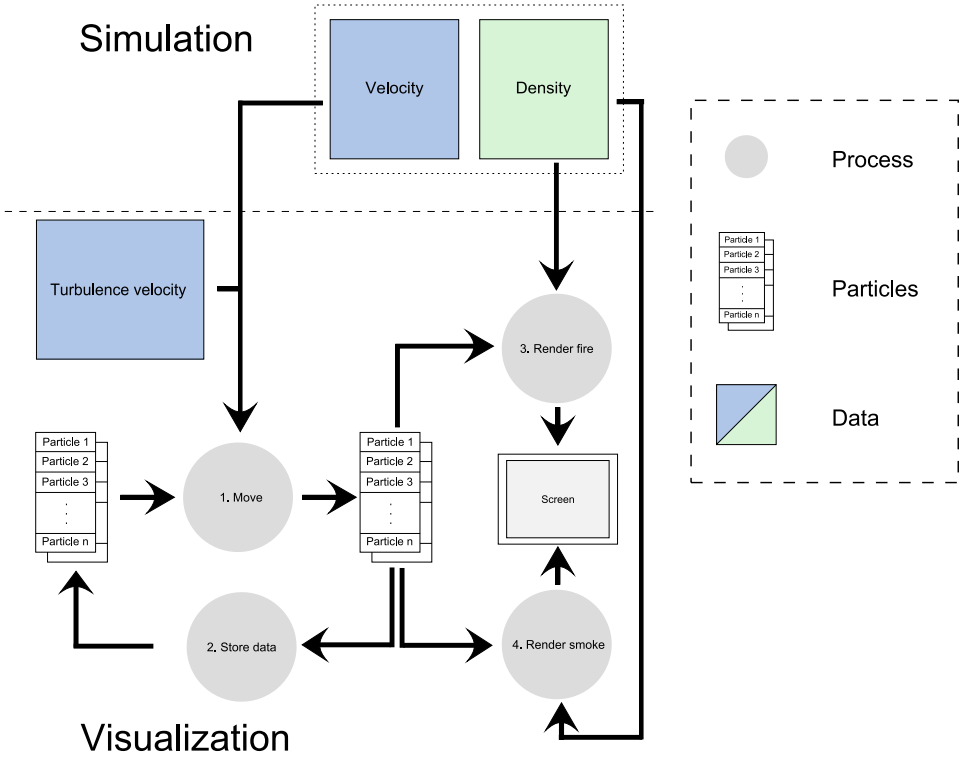

Figure 5.1: Outline of the visualization step

5.2 Particle movement

Every particle, i, is positioned in space at $\mathbf{x}_i$ and updated after each simulation step. The particles are considered massless, and move with the fluid velocity sampled from the simulation result and velocity from a turbulence model. The following equation shows how to calculate the new position, $\mathbf{x}_i^{new}$:

$$\mathbf{x}_i^{new} = \mathbf{x}_i + \Delta t \cdot (\mathbf{u}_{\mathbf{x}_i} + \mathbf{v}_{\mathbf{x}_i}) \cdot \omega_i \tag{5.1}$$

where Δt is the time step, $\mathbf{u}_i$ and $\mathbf{v}_i$ are the fluid velocity from the simulation step and the turbulence velocity, respectively. The fluid velocity is sampled as described in section 4.5, while the turbulence velocity is explained below.

The explosion simulation generates high velocities, especially during the first few moments of the simulation. If all particles were to be moved directly by this velocity, they would all be relocated to the other region of the explosion during the first few time steps. To avoid this, we introduce a movement factor, ω_i, which is assigned to each particle and create diversity in how far they move. ω_i can be thought of as a particle's inertia, but bear in mind that it has no actual physical meaning. Particles with a small ω_i have the tendency to dance around at the outer regions of the simulation, representing the turbulent and characteristic rolling motion of explosions. Particles with larger ω_i tend to slowly follow the velocity field, visualizing the rising buoyant plume. By choosing an appropriate distribution for randomly selecting the ω_i values for the particles, the amount of simulation detail the visualization algorithm is able to visualize can be maximized. Other approaches for moving particles exist such as the more physically correct method used in [FOA03] or Euler steps as used in [GRS06].

5.2.1 Turbulence

Explosions generate large amounts of turbulence, but such seemingly random fluctuations are not easily retrieved by solving the Navier Stokes equations alone. Consequently, scientific simulations often use turbulence modeling in addition to fluid simulations [Dav97]. The simulation step uses vortex particles to add additional rotational motion to the fluid. However, the cylindrical extrusion used when sampling from the 2D slice simulation causes symmetry regardless of simulation step turbulence. In addition to using vortex particles, which directly affect the fluid velocity in the simulation domain, a turbulence model is used in the visualization step. The motion of the particles is affected independently of the simulation velocity, namely by finding a value for $\mathbf{v}_i$ of equation 5.1. This way, not only turbulence is added, but symmetries introduced by the 2D slice sampling may be broken up.

The turbulence model is used to construct three dimensional velocity fields containing small-scale eddies (swirlings in the fluid). These velocity fields are periodic and may consequently be tiled in space and time. Thus, one turbulence field of three spatial dimensions is enough to define turbulence

anywhere in space. For a brief introduction to how these turbulence fields are generated, see section 5.2.2.

To be able to adjust the frequency of small-scale variation, the turbulence field, which is defined in the range [0,1], is scaled in each spatial direction by the scalars l_x, l_y, and l_z. Then, to let turbulence be defined all over space the turbulence field is set to repeat itself every l_x, l_y, l_z, also for each spatial direction. However, to find the turbulence velocity at a position in space, the coordinates need to be transformed to the range [0,1]. Any position $\mathbf{x}$ in space can be mapped to this range using the following equation:

$$p_i = \frac{x_i}{l_i} - \lfloor \frac{x_i}{l_i} \rfloor, \quad i \in \{x, y, z\} \tag{5.2}$$

Note that $\mathbf{x}$ may also contain negative components. The lower left rectangle of figure 5.2 represents a 2D turbulence field. The figure shows how three positions outside are mapped to their corresponding values inside the turbulence field. Note that the figure does not illustrate the scaling of values to the [0,1] range. Having found positions in the desired range, trilinear interpolation is used to find velocities in between grid velocity values.

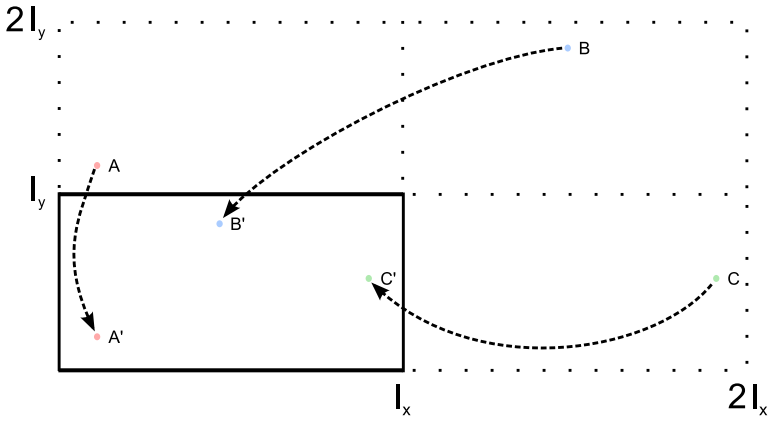

Figure 5.2: Three positions, A, B, and C, and their corresponding values, A', B', and C' inside the turbulence field

To allow turbulence to vary in the time domain, two turbulence velocity fields are generated, each of them still representing turbulence velocity all over space. The velocity fields are placed at separate points in time and linear interpolation is used to find velocities at intermediate times. The first field is defined at time $t = 0$ while the second is defined at the end of a period of length t_p. To find an interpolation parameter, $k \in [0, 1]$, we use the following equation:

$$k(t) = \begin{cases} \frac{t}{t_p} - \left\lfloor \frac{t}{t_p} \right\rfloor & \text{for even } \left\lfloor \frac{t}{t_p} \right\rfloor \\ 1 - (\frac{t}{t_p} - \left\lfloor \frac{t}{t_p} \right\rfloor) & \text{for odd } \left\lfloor \frac{t}{t_p} \right\rfloor \end{cases} \tag{5.3}$$

where t is the total simulation time. k will alternate smoothly between 0 and 1 as the value of t rises. The complete algorithm for sampling turbulence velocity given arbitrary values $\mathbf{x}$ and t is as follows:

- Find a position $\mathbf{p}$ from $\mathbf{x}$ and 5.2.
- Find velocities $\mathbf{v}_1$ and $\mathbf{v}_2$ by sampling from the two velocity field using trilinear interpolation at $\mathbf{p}$.
- Find k from the time, t, and equation 5.3.
- Find $\mathbf{v}$ by linearly interpolate between $\mathbf{v}_1$ and $\mathbf{v}_2$ using equation 5.4.

$$\mathbf{v} = \mathbf{v}_1(1-k) + \mathbf{v}_2 k \tag{5.4}$$

The final turbulence value, $\mathbf{v}$, sampled from a position $\mathbf{x_i}$, is used as $\mathbf{v}_{\mathbf{x}_i}$ in equation 5.1.

5.2.2 Creating turbulence velocity

We adopt the method proposed in [SF93] to generate a velocity field defined by an energy spectrum. Their method is general in the way that any valid energy spectrum, $E(k, \omega)$, may be used, where k is the length of a spatial frequency vector $\mathbf{k}$. Numerous energy spectrums can be found in the turbulence literature, describing various phenomena. An energy spectrum that has been successfully applied to the computer graphics domain is the Kolmogorov spectrum. By applying some simplifying assumptions of the fluid, one is able to describe how energy of turbulent motion propagates to higher frequencies (i.e smaller eddies). The equation for the Kolmogorov energy spectrum is shown in equation 5.5 and expresses the amount of energy at spatial frequencies $\mathbf{k}$ when energy is introduced at a frequency $k_{inertial}$ and propagated at the constant rate ε. k is the length of the vector $\mathbf{k}$.

$$E_K(k) = \begin{cases} 0 & \text{if } k < k_{inertial} \\ 1.5\varepsilon^{\frac{2}{3}} k^{-\frac{5}{3}} & \text{otherwise} \end{cases} \tag{5.5}$$

Note that equation 5.5 is not dependant on a temporal frequency, ω, thus representing steady-state turbulence. To allow the energy spectrum to vary with time, [SF93] modeled temporal frequency dependence by multiplying the spectrum with a temporal spread function that guarantees the conservation of kinetic energy. The spread function they used was a Gaussian with a standard deviation proportional to k:

$$G_k(\omega) = \frac{1}{\sqrt{2\pi}k\sigma} e^{-\frac{\omega^2}{2k^2\sigma^2}} \tag{5.6}$$

The result is the following equation for the energy at spatio-temporal frequencies $\mathbf{k}$ and ω:

$$E(k,\omega) = E_K(k)E_k(\omega)$$

[SF93] makes certain assumptions and performs an analysis of the cross-correlation of the velocity components (of the expected result), and its representation in the frequency domain that is beyond the scope of this thesis. Instead, we present the remaining equations needed to explain the turbulence field algorithm below. Cross-spectral density function are on the form

$$\Phi_{ij}(\mathbf{k},\omega) = \frac{E(k,\omega)}{4\pi k^4}(k^2\delta_{ij} - k_i k_j), \quad i,j \in \{1,2,3\} \tag{5.7}$$

where δ_{ij} is the Kronecker delta[1]. Also needed is the equations for a set of convolution kernels. Convolution in the spatio-temporal mirrors to multiplication in the frequency domain, simplifying these convolutions ($\hat{h}_{12} = \hat{h}_{13} = \hat{h}_{23} = 0$):

$$\begin{array}{lll} \hat{h}_{11} = \sqrt{\Phi_{11}} & & \\ \hat{h}_{21} = \frac{\Phi_{21}}{\hat{h}_{11}}, & \hat{h}_{22} = \sqrt{\Phi_{22} - \hat{h}_{21}^2} & \\ \hat{h}_{31} = \frac{\Phi_{31}}{\hat{h}_{11}}, & \hat{h}_{32} = \frac{\Phi_{32} - \hat{h}_{31}\hat{h}_{21}}{\hat{h}_{22}}, & \hat{h}_{33} = \sqrt{\Phi_{33} - \hat{h}_{31}^2 - \hat{h}_{32}^2} \end{array}$$

Note that all $\hat{h}_{ij}$ are functions of $\mathbf{k}$ and ω and that Φ is function 5.7, though the parameters are omitted for clarity in the equations above.

[1] The Kronecker delta is defined as follows:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

As stated previously, we only intend to create velocity fields of three spatial dimensions, thus we always set the temporal frequency, ω, to zero. This fact simplifies the algorithm proposed by [SF93], as well as equation 5.6. However, the general idea remains. It is to fill the frequency domain with random complex numbers and apply the convolution terms $\hat{h}_{ij}$, followed by an inverse Fourier transform. To further simplify the algorithm, though not necessary, the number of samples in each spatial direction is assumed to be N. To ensure that the resulting velocity field is real valued, the following symmetry must be satisfied: $\hat{\mathbf{u}}_{i,j,k} = \hat{\mathbf{u}}^*_{N-i,N-j,N-k}$, where the indices are taken modulo N and $*$ is the complex conjugate operator. In the special case where the indices on both sides are equal, the imaginary parts of $\hat{\mathbf{u}}$ is set to zero.

The algorithm for filling in the values is based on the symmetry condition, in the way that it is only necessary to calculate half the values in each spatial direction and derived the remaining ones:

for i, j, k in $\{0, 1, 2, \ldots, N/2\}$ do
  compute $\hat{\mathbf{u}}_{i,j,k}$, $\hat{\mathbf{u}}_{i,j,N-k}$, $\hat{\mathbf{u}}_{i,N-j,k}$, and $\hat{\mathbf{u}}_{i,N-j,N-k}$
  $\hat{\mathbf{u}}_{N-i,N-j,N-k} = \hat{\mathbf{u}}^*_{i,j,k}$
  $\hat{\mathbf{u}}_{N-i,N-j,k} = \hat{\mathbf{u}}^*_{i,j,N-k}$
  $\hat{\mathbf{u}}_{N-i,j,N-k} = \hat{\mathbf{u}}^*_{i,N-j,k}$
  $\hat{\mathbf{u}}_{N-i,j,k} = \hat{\mathbf{u}}^*_{i,N-j,N-k}$
end

for i, j, k in $\{0, N/2\}$ do
  set imaginary part of $\hat{\mathbf{u}}_{i,j,k}$ to zero
end

To compute the elements of $\hat{\mathbf{u}}_{i,j,k}$, generate a random complex variables $X_m = r_m e^{2\pi i \theta_m}, (m \in \{1, 2, 3\})$, where r_m and θ_m are real valued random numbers with Gaussian and uniformly distribution, respectively. Finally, calculate the three spatial components of $\hat{\mathbf{u}}_{i,j,k}$:

$$\begin{aligned} \hat{u}_1 &= \hat{h}_{11}((i,j,k),0)X_1 \\ \hat{u}_2 &= \hat{h}_{21}((i,j,k),0)X_1 + \hat{h}_{22}((i,j,k),0)X_2 \\ \hat{u}_3 &= \hat{h}_{31}((i,j,k),0)X_1 + \hat{h}_{32}((i,j,k),0)X_2 + \hat{h}_{33}((i,j,k),0)X_3 \end{aligned} \tag{5.8}$$

It is advantageous to precalculate the velocity fields and save them for use in the real-time application, as the generation process takes some time.

5.3 Particle Rendering

After the movement step, the particles are rendered as screen-aligned quads. The particles' positions are transformed from their local coordinate system to perspective camera coordinates by multiplying each position by a world-view-projection matrix. Next, the a quad is extruded from each particle's position aligning it to the view plane. The size of the quads are based on predefined constant, but the perspective projection particles look smaller as they are located further away from the camera. Furthermore, the appearance of hot burning gas is highly turbulent and rather small particles tend to look the most realistic. Smoke on the other hand, appears to be smoother and less turbulent, thus could be visualized with larger particles that blend into each other.

The color and visibility calculation of the two kinds of particles are different, but parts of both are based on black-body radiation. Black-body radiation is explained first, followed by fire and smoke color calculation.

5.3.1 Black-Body Radiation

The color of emitted light from hot gas is approximated by a black-body radiation model. Black bodies absorb all light that falls onto them, thus they are considered sources of purely thermal radiation. Planck's equation for the intensities of radiation based on the object's temperature in Kelvin, T, and wave lengths of the radiance, λ, is shown in 5.9.

$$E(\lambda, T) = \frac{2hc^2}{\lambda^5(e^{\frac{hc}{\lambda kT}} - 1)} \tag{5.9}$$

where h, c, and k are the Planck constant, speed of light, and Boltzmann constant, respectively. Colors on computer monitors consist of intensities of red, green, and blue, thus the wave lengths of these colors can be used with equation 5.9 to find color intensities $E_{red}, E_{green}, E_{blue}$ as a function of temperature.

The resulting intensity values have a very high range, not suitable for ordinary computer monitors, which uses intensity values in the range [0, 1]. To map color intensities onto the desired range, the exponential mapping from [Mat97] is used:

$$I = 1 - e^{\frac{-E}{E_{average}}} \tag{5.10}$$

where E is the original intensity of either red, green, or blue, and $E_{average}$ is a constant that controls the average intensity. The result is new intensities, I, that fall into the range $[0, 1\rangle$.

The calculation of color intensities for both red, green, and blue for fluid temperature is a process that is too expensive to be computed each frame, nor is it necessary. The calculation results would remain the same during the complete simulation, either way, so precalculating these values is the most sensibly action. Color values for various temperatures in a predefined temperature range are stored in a lookup table for easy retrieval.

5.3.2 Fire color calculation

The color of the fire particles are based on black-body radiation, as explained above, while the intensity of the color is specified by the amount of exhaust gas. Thresholds for the temperature and exhaust gas levels where fire is fully visible are predefined. Equation 5.11 shows how to calculate the color of a particle positioned at $\mathbf{x}_i$:

$$C_{fire} = B(T_{\mathbf{x}_i}) \cdot Step(0, T_\alpha, T_{\mathbf{x}_i}) \cdot Step(0, E_\alpha, E_{\mathbf{x}_i}) \tag{5.11}$$

where B is the lookup function into the precalculated black-body radiation table, while T_α and E_α is the threshold levels described above. These are not used as strict thresholds, however, but are used to define where the color intensity starts to decrease.

$$Step(low, high, x) = \begin{cases} 0 & \text{if x} < \text{low} \\ 1 & \text{if x} > \text{high} \\ -2(\frac{x-low}{high-low})^3 + 3(\frac{x-low}{high-low})^2 & \text{else} \end{cases} \tag{5.12}$$

Step is a smooth stepping function, shown by equation 5.12, that returns zero if x is below low, one if it is above $high$, and a smoothly interpolated value in the range [0,1] in-between. It ensures smooth visual transitions between areas of fire and smoke.

$$C_{dst}^{new} = C_{src} + C_{dst} \cdot (1 - C_{src}) \tag{5.13}$$

Next, the fire particles are drawn to the screen blended with each other and possible existing content using the blending function described in equation 5.13. C_{dst}^{new} is the new color of the screen, C_{src} is the color of the particle

being drawn, and C_{dst} is the old color of the screen. An advantage of this type of blending is its invariance of particle draw order [SR06], hence there is no need for expensive sorting algorithms. Note that the intensity calculation of the fire color is embedded in equation 5.11 because of the blending function used. Colors with low intensity will not affect the result as much as color with higher intensities.

5.3.3 Smoke color calculation

The smoke color and intensity are used separately. The intensity is used as the final color's α-value (opacity) instead of adjusting the color directly as with the fire color calculation. This is to allow dark, dense smoke to conceal what is behind it. The actual color is the sum of a predefined $Grey$ color and a small portion, ϵ, of a color sampled from the black-body radiation table, as shown in the following equation:

$$C_{smoke} = Grey + \epsilon \cdot B(T_{\mathbf{x}_i}) \tag{5.14}$$

The smoke color's α-value is calculated by equation 5.15. The assumption that led to that equation is that what we perceive as smoke is actually exhaust gas that has cooled down.

$$\begin{aligned} A_{smoke} = & k \cdot \\ & (1 - Step(0, T_{max}, T_{\mathbf{x}_i})) \cdot \\ & Step(E_{min}, E_{max}, E_{\mathbf{x}_i}) \cdot \\ & Step(t_{start}, t_{full}, t_{sim}) \end{aligned} \tag{5.15}$$

The second factor of equation 5.15 makes sure that smoke is not visible where ever the temperature, $T_{\mathbf{x}_i}$, is below a certain threshold, T_{max}, but smoothly interpolated to zero as the temperature decreases. Similarly, the third factor ensures that smoke is not visible if the exhaust gas level is below E_{min}. Also, we adjust the maximum exhaust level, T_{max}, where higher levels no longer affect the smoke density. The constant, k is used to control the overall density of the smoke. Note that the assumption that smoke is exhaust gas that has *cooled down* is not fully complied with yet. The simulation does not provide enough information to facilitate such a condition, nor is it necessary. We have found that a simple approximation based on the cumulative simulation time t_{sim} is sufficient, and fade the smoke in after a short predefined period of time. The final factor of the equation takes care of this by the use of the *Step* function. The smoke starts to fade in at t_{start} and is fully visible at t_{full}.

Finally, the particles are blended on top of each other and the already drawn fire using the following blend function [Hol03]:

$$C_{dst}^{new} = A_{src} \cdot C_{src} + (1 - A_{src}) \cdot C_{dst} \tag{5.16}$$

where A_{smoke} is used as C_{src} to control the opacity. Particles being rendered last affect the final appearance of the explosion more than ones being drawn previously. Hence, this blending function depends on the draw order of the particles to be correct, and the particles should optimally be sorted based on their distance to the camera and rendered in a back-to-front order. We omit such sorting because it is a time consuming operation and would counteract the real-time requirement. Furthermore, as discussed more thoroughly in chapter 7, the visual results do not seem to suffer substantially either.

5.3.4 Rendering animated texture splats

As previously stated, the particles are rendered as screen-aligned quads. To create more low level detail and to reduce the number of particles needed, they are textured. The textures are not static like in [WLMK02, Hol03], but are rather animated introducing additional variation to the explosion (such as the smoke from [Mic06]). The animations used are generated based on a static texture splat and animations generated by the Kolmogorov turbulence algorithm explained in section 5.2.2. The static texture splats used are shown in figure 5.3. The left splat is used for the generation of fire, while the one to the right is used for smoke texture animation.

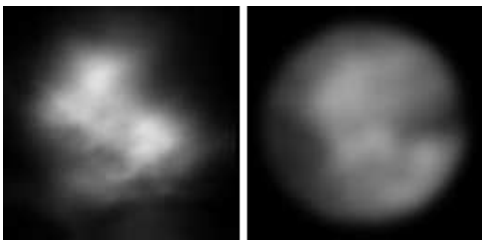

Figure 5.3: Fire (left) and smoke (right) texture splats that are used as animation bases

To create the animations used for the texture splats, the 3D Kolmogorov velocity field is used to generate sequences of frames that subsequently are blended with the static texture splats. Each frame in the turbulence sequence is based on the lengths of the velocities in each slice of the 3D turbulence field. Because the 3D Kolmogorov velocity field is tileable in space,

the produced animation can be looped seamlessly. This process is performed in a stand-alone application, which is explained in more detail in chapter 6.

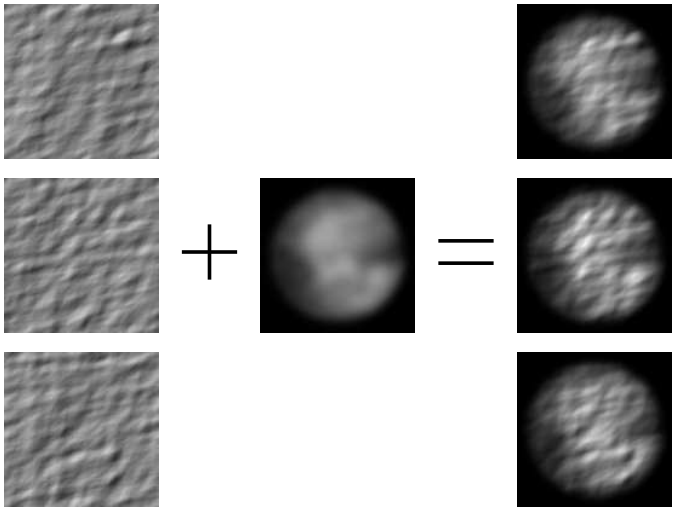

Figure 5.4: Three frames of turbulence (left) combined with a texture splat (middle) produces three frames of the animated texture (right)

Figure 5.4 shows how three frames from the turbulence field are combined with a texture splat to produce frames for a smoke particle animation. Notice how small scale variations are introduced in the rightmost pictures. A small disadvantage of using this technique, is that the animations appear as waves moving in a fixed direction. If all particles have the same orientation, an joint wave-like motion will appear over the entire explosion. We solve this minor problem by giving each particle a random, but fixed orientation. This way each particle's wave-like motion blends into the other's removing unwanted joint motion.

5.4 The complete algorithm

The following algorithm shows the complete steps taken for moving and rendering the particles:

1. Move all fire and smoke particles based on equation 5.1 and store the new positions.

    - Find $\mathbf{u}_i$ from the velocity field from the simulation step, based on the particle's position.

- Find $\mathbf{v}_i$ from the turbulence velocity field, using the algorithm described in section 5.2.1.

2. Transform the particle positions from their local coordinate system to view coordinates based on the position of the camera.

3. Render fire particles to the screen based on their view coordinate position.

   - Calculate the particle's color using equation 5.11 based on the density sampled from the simulation results.
   - Increase the animation frame number, based on a fixed animation speed.
   - Orient the particle based on its random orientation.
   - Use the blending function described by equation 5.13.

4. Render smoke particles to the screen based on their view coordinate position.

   - Calculate the particle's color using equation 5.14 based on the density sampled from the simulation results.
   - Increase the animation frame number, based on a fixed animation speed.
   - Orient the particle based on its random orientation.
   - Use the blending function described by equation 5.16.

5.5 Summary

We have presented a two-component particle systems for visualizing the result of the explosion simulation. All particles are moved by the velocity sampled from the simulation, combined with independent turbulence velocity from a Kolmogorov turbulence field.

The particles are rendered as screen-aligned quads, textured with a turbulence animation based on the Kolmogorov spectrum. Fire and smoke particles use different color and visibility calculations. The fire is rendered first, followed by smoke being blended on top.

Chapter 6
Implementation

This chapter describes how the simulation and visualization of explosions using the GPU are implemented. The implementation is based on the simulation algorithm presented in section 4.4 and the visualization algorithm presented in section 5.4. Also, the simulation makes use of an implementation of the fluid solver presented in section 2.2.4. Additionally, two stand-alone applications are implemented. The purpose of the first application is to create the Kolmogorov turbulence texture as explained in 5.2.2. The second application creates an animated texture based on a texture splat and a Kolmogorov turbulence texture. This texture is used to render the particles as explained in 5.3.4.

The rest of the chapter is organized as follows: First, an overview of the implementation is given, where the different modules into which the implementation is divided and their connections are described. Next, each of the modules will be described in more details, starting with a simple framework for performing general-purpose computation on the GPU. The implementation of the fluid solver is described next, before the implementations of the explosion simulation and visualization are presented. Finally, each of the two stand-alone applications is described.

6.1 Overview

The implementation is divided into modules, and figure 6.1 gives an overview of these modules. The first module is the *GPU Computational Framework*. This module contains functionality whose purpose is to ease the task of performing general-purpose computation on the GPU, and it is used by all the other modules. Next, the *Fluid Solver* module contains an implementation of a general-purpose fluid solver. This fluid solver is used in the *Explosion Simulator* module. Finally, the *Explosion Visualizer* module visualizes the results from the *Explosion Simulator* by the use of two particle systems; one for visualizing the fire, and one for visualizing the smoke.

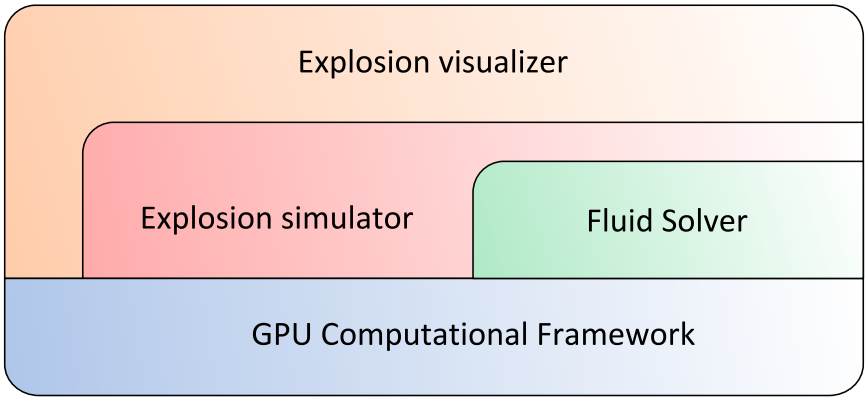

Figure 6.1: Module overview

6.2 GPU Computational Framework

As described in section 2.3, the GPU can be used to perform general-purpose computations. This includes executing shader programs, and performing computations on textures. This may be a cumbersome task, and the purpose of this framework is to make this functionality easier available to the user.

6.2.1 Class overview

Figure 6.2 shows a selection of the most important classes in the framework along with an overview of the most important methods of each class. The functionality and suggested usage of each class is described next.

6.2.1.1 TechniqueOperation

The framework support shader programs written in the Hlsl language, which is one of the available high-level shader languages. The various shader programs are written in effect files, and each file may contain one or more techniques. Each technique can contain a vertex shader, a geometry shader, and a pixel shader, and these shaders programs are typically performed by invoking a technique. Invoking a technique often require some amount of code, and the purpose of the `TechniqueOperation` class is to ease this task.

Before the `TechniqueOperation` class can be used, an instance has to be created by passing a few required parameters to the constructor. These parameters include the effect file in which the technique is located, the name of the technique itself, and a reference to a texture store. This texture store

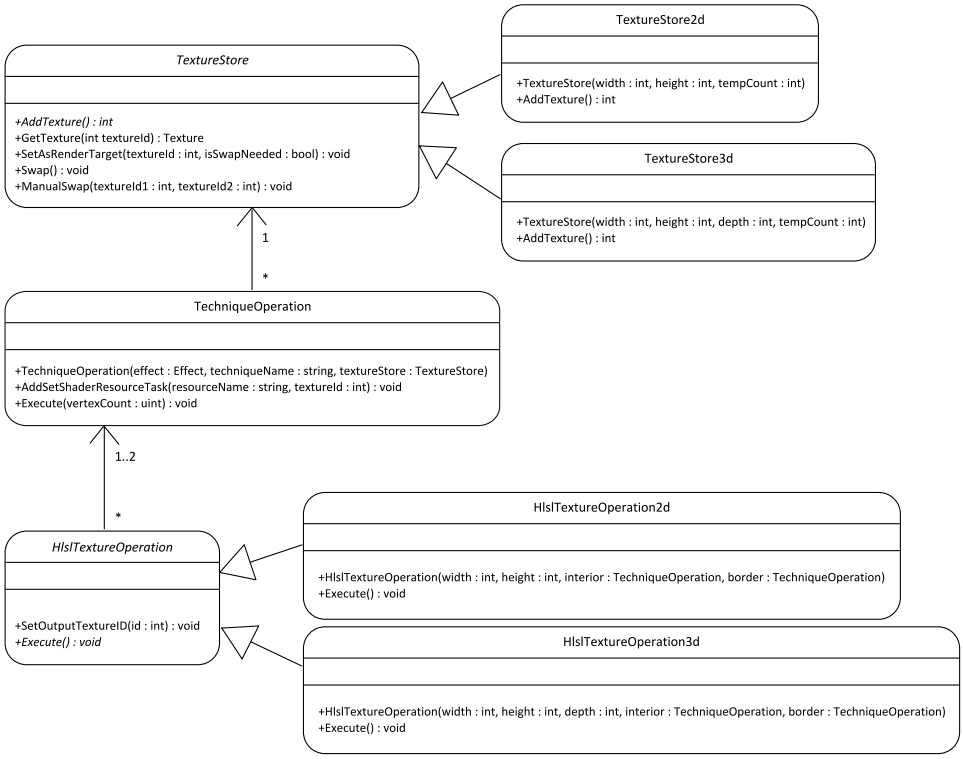

Figure 6.2: Overview of the most important classes and methods in the GPU computational framework

is used by the `AddSetShaderResourceTask` method, which will be explained shortly. Once an instance of the class exists, the technique can be invoked by calling the `Execute` method.

Textures are often used when performing general-purpose computation on the GPU. They are maintained by a texture store, and they are represented by identifiers. In an effect file, a texture is referenced as a shader resource, and one often wants to explicitly set a shader resource to a texture every time a shader program is invoked. This is done by the use of the `AddSetShaderResourceTask` method. This method takes as parameters the name of the shader resource that will be set to the given texture, as well as the identifier of the texture itself.

6.2.1.2 TextureStore

The `TextureStore` class is an abstract class used to maintain the textures that are used in the computations. A texture store can contain either two-

dimensional or three-dimensional textures, and either the `TextureStore2d` or the `TextureStore3D` subclass is used depending on what kind of texture is needed. Also, a texture store can only contain similar sized textures. This size is passed to the constructor when an instance of the texture store is created.

Once a texture store instance is available, textures can be added by the use of the `AddTexture` method, which returns the identifier of the newly created texture. This identifier can then be passed to the `GetTexture` method to retrieve a reference to the actual texture.

When textures are to be used for output, they are set as render targets. This can be done by the use of the `SetAsRenderTarget` method to which the identifier of the output texture is passed as parameter. Sometimes it is desirable to use a texture both as input and output at the same time. This can be achieved by using a temporary texture as the render target, the given texture as input, and then swap the identifiers of the temporary and the given texture after the render operation is done. This process is made easier by the use of the `SetAsRenderTarget` method. In addition to the texture identifier, a boolean have to sent as parameter as well, specifying if the given texture is going to be used as input as well. If so, a temporary texture is used as render target instead. Then, after the render operation is complete, the identifiers are swapped simply by calling the `Swap` method.

Finally, the `ManualSwap` method can be used to swap any two given textures by passing the identifiers of the textures as parameters.

6.2.1.3 HlslTextureOperation

The `HlslTextureOperation` class is an abstract class that is used to perform operations on a texture. The shader programs that are to be used are represented by instances of the `TechniqueOperation` class. One technique operation is used for the interior cells of the texture, and one technique operation is used for the boundary cells[1]. Both operations are optional. This way, the user can decide whether to perform calculations on the interior cells, the boundary cells, or both.

The technique operations are passed as parameters to the constructor when an instance of the class is to be created, along with the size of the output texture. The desired output texture is set by the use of the `SetAsRenderTarget` method in the respective texture store. If the output texture is used as input as well, the swapping functionality in the texture store is utilized automatically. Once a texture is set as render target, the texture operation

[1]The difference between boundary cells and interior cells is explained in 4.1.1

is performed by the use of the `Execute` method.

6.2.1.4 HlslTextureOperation2d

The `HlslTextureOperation2d` class is a specialization of the `HlslTextureOperation`, and is supposed to be used for operations on two-dimensional textures. When the `Execute` method is invoked, the viewport is set to match the size of the output texture, and a single quad is drawn. The resulting pixels correspond to the pixels in the output texture, and new values are calculated by the execution of pixel shader programs represented by the interior and boundary technique operations.

6.2.1.5 HlslTextureOperation3d

Operations on three-dimensional textures are performed by the use of the `HlslTextureOperation3d` class. Here, when the `Execute` method is invoked, the viewport is set to match the width and height of the texture, and a quad is drawn for every depth value. The resulting pixels from each quad corresponds to a slice in the three-dimensional output texture, and the targeting slice is selected by setting the render target array index to the corresponding depth value. Finally, the pixel shader programs represented by the interior and boundary technique operations are executed to calculate the new texture values.

6.3 Fluid solver

The general fluid solver that is implemented is based on the Stable Fluids method that was described in section 2.2.4. It is implemented using shader programs on the GPU, and makes use of the GPU computational framework. It is divided up into two different parts. The first part contains the velocity specific part of the fluid solver, as explained in section 2.2.4.2. The second part implements the density specific part that was described in section 2.2.4.3. Both these implementations will be described in further details in the following sections. First, however, the various textures that are used in the fluid solver will be described.

6.3.1 Textures

The calculations that are performed in the fluid solver are performed on velocity and density fields. To be able to perform the calculations on the GPU, these fields first have to discretized and represented as textures. Since a texture can contain up to four color channels, up to four density fields can be packed into and represented by a single texture. Similarly, both two- and three-dimensional velocity fields can be represented by a single texture by packing the various components of the velocity field into the various channels of the texture. Packing several fields into a texture in this fashion greatly decreases the number of necessary textures, and thus also the number of rendering passes.

Most of the textures that are used by the fluid solver has to be supplied by the user. An overview of these texture and the contents of each channels is shown in table 6.1. Also, a few textures are used internally by the fluid solver in the calculation of the intermediate steps. These textures are shown in table 6.2.

Texture	Red channel	Green channel	Blue channel	Alpha channel
Velocity	The x-component.	The y-component.	The z-component (3d only).	
VelocityForces	The x-component.	The y-component.	The z-component (3d only).	
Density	The amount of density *a*.	The amount of density *b*.	The amount of density *c*.	The amount of density *d*.
DensityChanges	The amount of density *a* change.	The amount of density *b* change.	The amount of density *c* change.	The amount of density *d* change.
DivergenceForces	The amount of divergence force.			
Obstacles	Whether or not there is an obstacle.			

Table 6.1: Overview of textures

In table 6.1, the *Velocity* texture contains the vector components of the velocity field, the *VelocityForces* texture contains the vector components of the velocity force field, the *Density* texture contains the density amounts of up to four density fields, the *DensityChanges* texture contains the amounts of change of up to four density fields, and the *DivergenceForces* texture contains the amount of force that is to be applied on the divergence. Finally, the `Obstacles` texture specifies whether or not there is an obstacle at a given position.

In table 6.2, the *Divergence* texture contains the divergence amounts of a divergence field, the *Pressure* texture contains the pressure amounts of a pressure fields, and the *DiffusedDensity* texture contains the density amount of a density field during the diffusion step.

The usage of and calculations performed on each texture will be described

Texture	Red channel	Green channel	Blue channel	Alpha channel
Divergence	The amount of divergence.			
Pressure	The amount of pressure.			
DiffusedDensity	The amount of density a after diffusion.	The amount of density b after diffusion.	The amount of density c after diffusion.	The amount of density d after diffusion.

Table 6.2: Overview of textures

in the following sections.

6.3.2 Class overview

The implementation of the fluid solver module is divided into a density specific and a velocity specific part. The classes `DensityOperation2d` and `DensityOperation3d` implements the density specific operations for two- and three-dimensional fluid flow respectively, whereas the classes `VelocityOperation2d` and `VelocityOperation3d` implements the operations that are performed on the velocity values.

6.3.2.1 VelocityOperation2d and VelocityOperation3d

The two `VelocityOperation` classes implement the velocity part of the Stable Fluids method, partly based on the algorithm that was described in section 2.2.4.2. Both advection, projection, and adding of external force is implemented, though the viscosity part is omitted for reasons explained in section 4.2.1.

External forces
External forces are added in the `AddVelocityForces` shader program. This shader program takes two textures as parameters: a texture containing all the forces, and a texture containing all the velocities to which the forces are to be added. For every pixel being processed, a force $\mathbf{f}$ is fetched from the force texture. The force is multiplied with a timestep Δt before it is added the the current velocity value $\mathbf{u}$ from the velocity texture. Both the force and velocity value is fetched from the position that corresponds to the pixel being processed. The entire calculation that is performed on every pixel is shown in equation 6.1.

$$\mathbf{u}' = \mathbf{u} + \mathbf{f}\Delta t \tag{6.1}$$

Advection
The advection part is implemented in the `Advection` shader program. Two

textures are taken as parameters: a velocity texture containing the velocity field, and a texture containing the field that is to be advected by the velocity field. For velocity advection, the velocity field is self-advecting, and thus the same texture should be used for both parameters.

As explained in section 2.2.4.2, the new values that result from the advection step are set to the values at the positions the values would have ended up at if they had been advected backwards in time. Also, since these positions can be located in between cells, the new values have to be calculated by bi-linearly interpolating between the values from the four most nearby cells.

In practice this is done by first fetching the velocity value $\mathbf{u}$ from the velocity texture. This velocity value is multiplied by a timestep Δt before it is subtracted from the current position $\mathbf{x}$. This resulting position $\mathbf{x}'$ is the position at which the value would have ended up if it was advected backwards. The calculation is shown in equation 6.2.

$$\mathbf{x}' = \mathbf{x} - \mathbf{u}\Delta t \tag{6.2}$$

Now, the new value is found by fetching the value at this position. Bilinear interpolation between the values from the four most nearby values is done automatically by the GPU by the use of linear texture sampling.

Texture fetches at the resulting position $\mathbf{x}'$ can lead to texture fetches outside the boundary of the texture and these fetches thus have to be clamped to the texture boundary. This is done automatically by the GPU by correctly setting the sampler state.

Projection

The purpose of the projection part is to remove any divergence that may have occurred in the other steps of the algorithm, as described in section 2.2.4.2. This involves calculating the pressure field that corresponds the the velocity field and then subtract its gradient from the velocity field.

This behavior is implemented in the `CalculateDivergence`, `CalculatePressure`, and `SubtractDivergence` shader programs. First, the `CalculateDivergence` shader program is used to compute the divergence of the velocity field, which is stored in a temporary divergence texture. To support thermal expansion, a divergence force texture can be used, allowing this temporary divergence texture to be modified before the pressure is calculated.

Next, the temporary divergence texture is used by the `CalculatePressure` shader program when it computes the pressure field. Computing this pressure field requires solving the Poisson equation shown in equation 2.12. The equation is discretized into a set of linear equations that are solved using a

Jacobi solver. The Jacobi solver needs several iterations to find a sufficient solution. The calculation that is done in each Jacobi iteration step is shown in equation 6.3. Here, b_i is the divergence of the corresponding velocity from the temporary divergence texture, α is set to 1, while β is set to number of neighbours (4 for the 2d simulation and 6 for the 3d simulation).

$$x_i^n = \frac{\sum_{j \in neighbours(x_i)} \left(x_j^{n-1}\right) - \alpha \cdot b_i}{\beta} \tag{6.3}$$

Finally, once the pressure field is calculated and stored in a temporary pressure texture, it's gradient is calculated and subtracted from the velocity field in the `SubtractDivergence` shader program.

6.3.2.2 DensityOperation2d and DensityOperation3d

The density part of the Stable Fluids method, as described in section 2.2.4.3, is implemented in the two `DensityOperation` classes. This includes both the sources and dissipation, diffusion, and advection part of the algorithm.

Sources and dissipation
Sources are added and the densities are dissipated in the `AddDensitySources` shader program. This shader program takes two textures as parameters: a texture containing four densities, and a texture containing the sources that are to be added. First, a vector containing four source values $\mathbf{s}$ is fetched from the source texture. Then, to make dissipation of the densities possible, a vector of four density values $\mathbf{d}$ is fetched from the density texture, and multiplied with a vector of four dissipation constants α_d. If no dissipation is desired, the dissipation constant is set to zero. The resulting vector is then subtracted from the source vector, and the result is then multiplied with a time step Δt before it is added to the density value vector $\mathbf{d}$. The calculation is summarized in equation 6.4.

$$\mathbf{d}' = \mathbf{d} + (\mathbf{s} - \alpha_d \mathbf{d})\,\Delta t \tag{6.4}$$

Diffusion
The diffusion part is implemented in the `Diffusion` shader program. This shader program takes one texture as parameter, namely a density texture representing up to four density fields.

Diffusion is solved implicitly using equation 2.20. This equation results in a set of linear equations that are solved using a Jacobi solver, similar to the

projection step. The calculation that is done in each Jacobi step uses the same equation, namely equation 6.3. Here, x_i^n is a cell from the resulting density field from the n'th iteration, b_i is a cell from the old density field as it was before first iteration of the Jacobi solver, while α and β are calculated using equations 6.5 and 6.6 respectively.

$$\alpha = \frac{1}{\kappa \Delta t} \tag{6.5}$$

$$\beta = n + \alpha \tag{6.6}$$

Here, κ is the diffusion constant, Δt is the timestep, and n is set to the number of neighbours (4 for the 2d simulation and 6 for the 3d simulation).

Advection
Advection of density values is similar to advection of velocity values, and is in fact implemented using the same shader program, namely the `Advection` shader program. The only difference is which parameters are used. The first texture parameter still represents the velocity field, whereas the second texture parameter now represents up to four density fields that are to be advected by the velocity field. Invoking the shader program with these parameters results in the densities being advected instead of the velocities, as in the velocity part.

6.3.3 Boundary conditions

Two different approaches for handling boundary conditions have been implemented. In the 2D slice implementation only boundaries along the edges of the simulation domain are allowed. A border of cells along the edges are treated separately from the internal cells. As long as these border cells are handled in the proper way, all internal cells may use the same pixel shader that simply samples from its neighboring cells when it needs to.

In the full 3D implementation, the boundaries are handled in a more general fashion. Similarly to [NVi07], a 3D texture is initialized with values that represent whether a certain cell is a boundary cell or not. This allows the user to specify arbitrary internal boundary conditions. The complexity of the pixel shader code is greatly increased, as it needs to account for the fact that each single cell may be a boundary cell.

More specifically, the pixel shader code first checks whether or not its target cell is a boundary cell and chooses the correct branch. Then, for each relevant neighbor it checks whether those cells are boundary cells or not as

well. Each of these checks may yield a new branch operation that treats the boundary condition correctly.

The actual numerical operations that are performed for these boundary cases are explained in section 4.3.

6.4 Explosion simulation

The explosion simulation is implemented based on the algorithm described in section 4.4. The implementation makes use of the fluid solver described in section 6.3, and the main objective is to calculate the various textures that the fluid solver takes as input.

6.4.1 Textures

The textures that the fluid solver operates on and expects as input is shown in table 6.1. All these textures are created at the beginning of the explosion simulation, and they are passed along to the fluid solver once every frame. The densities that are used in the explosion simulation are the exhaust gas, the fuel gas, and the temperature. These are stored as densities a, b, and c respectively.

Only three of the textures are updated every frame in the simulator. These are the *VelocityForces*, the *DensityChanges*, and the *DivergenceForces* textures. The *Velocity* and *Density* textures are updated by the fluid solver based on these three textures. The `Density` texture is also modified directly at the start of the simulation to include the initial fuel and spark, as will be explained in section 6.4.2.1. Also, the `Obstacles` texture is modified at the beginning of the simulation if internal boundaries are to be included. A white color is rendered to obstacle cells, whereas a black color represents a non-obstacle cell.

6.4.2 Class overview

The explosion simulation in two and three dimension is implemented in the `ExplosionSimulator2d` and `ExplosionSimulator3d` classes respectively. The `VelocityOperation` and `DensityOperation` classes from the fluid solver are used to model the actual flow of the explosion. Also, the velocity forces, density changes, and divergence forces that are required by the fluid solver is implemented in the `CalculateVelocityForcesOperation`, `CalculateDensityChangesOperation`, and `CalculateThermalExpansionOperation`

classes respectively. All of these classes exist in two versions; one for the 2d simulation and one for the 3d simulation.

6.4.2.1 ExplosionSimulator2d and ExplosionSimulator3d

The two `ExplosionSimulator` classes implement the explosion simulation algorithm that was presented in section 4.4. Since the initial fuel and spark are necessary for the whole simulation to take place, these initial conditions are established first by rendering them into the density texture. This is done by the `RenderFuel` and `RenderSpark` shader programs respectively. Next, step 1 and 2 of the algorithm is implemented in the two `CalculateVelocityForcesOperation` classes, step 3 is implemented in the two `CalculateDensityChangesOperation` classes, and step 4 is implemented in the two `CalculateThermalExpansionOperation` classes. These classes will be described in more detail in the following sections.

Once both the velocity force texture, the density change texture, and the divergence force texture have been computed, they are sent as input to the fluid solver along with the velocity and density textures. The fluid solver then updates both the velocity and density textures.

6.4.2.2 CalculateVelocityForcesOperation2d and CalculateVelocityForcesOperation3d

The two `CalculateVelocityForcesOperation` classes make use of the `CalculateVelocityForces` shader program to calculate the velocity forces that result from buoyancy, gravity, and vorticity confinement. The buoyancy and gravity forces are calculated according to the equations 4.7 and 4.8 respectively, whereas the vorticity confinement is calculated based on equation 4.11.

The forces resulting from the vortex particles are added to the velocity forces texture in the `CalculateVortexParticleForces` shader program. The force induced from each vortex particle is calculated using equation 4.16. The actual movement of the vortex particles is performed in the `MoveVortexParticles` shader program, and the new positions are calculated using equation 4.13.

6.4.2.3 CalculateDensityChangesOperation2d and CalculateDensityChangesOperation3d

The changes to the density fields are a result of combustion and dissipation. These density changes are calculated in the `CalculateDensityChanges` shader program. This shader program takes the density texture as input, and writes the result to the density changes texture.

The density changes due to combustion affects both the fuel, exhaust gas, and temperature, and the changes to each of these are calculated using equations 4.17, 4.18, and 4.19 respectively. The changes due to dissipation is calculated using the dissipation part of equation 4.5.

6.4.2.4 CalculateThermalExpansionOperation2d and CalculateThermalExpansionOperation3d

Thermal expansion will be used as divergence forces in the velocity fluid solver step. The calculation makes use of equation 4.20, and is implemented in the `CalculateThermalExpansion` shader program. The results are written to the divergence force texture.

6.5 Explosion visualization

The visualization of the explosion is implemented based on the algorithm described in section 5.4. Particles are used to visualize the explosion, and the movement and appearance of the particles are based on the velocity and density fields from the explosion simulation. In addition to the actual explosion, if any internal boundaries are used, they are rendered as well. Also, since the generation of the 3D Kolmogorov turbulence fields and the animated textures are computationally demanding and can be pre-generated, stand-alone applications have been implemented to aid in this task.

6.5.1 Textures

The explosion visualizer uses several textures, and an overview of the textures that are used is given in table 6.3. Here, the `Velocity`, `Density`, and `Obstacles` textures are passed along from the explosion simulator, and contains the velocity field, the density field, and the obstacles, respectively. The `ColorLookUpTexture` texture contains a range of 256 precalculated color values that are used to calculate the color of a particle. The `FireParticleTexture` and `SmokeParticleTexture` textures contains pre-generated animated

textures used to render the fire and smoke particles respectively. Finally, the `TurbulenceTexture1` and `TurbulenceTexture2` contains the two 3D Kolmogorov turbulence fields that are used to add additional turbulence.

Texture	Description
Velocity	The vector components of the velocity field.
Density	The amounts of exhaust gas, fuel gas, and temperature.
Obstacles	The internal boundaries that are used, if any.
ColorLookUpTexture	A range of 256 color values based on black-body radiation.
FireParticleTexture	The animated fire particle.
SmokeParticleTexture	The animated smoke particle.
TurbulenceTexture1	The first 3D Kolmogorov turbulence texture.
TurbulenceTexture2	The second 3D Kolmogorov turbulence texture.

Table 6.3: Overview of textures

6.5.2 Class overview

The visualization is divided into an explosion and an obstacle part. The `ParticleVisualizer` class implements a particle system used to visualize the actual explosion, whereas the `ObstacleVisualizer` class is used to visualize the internal boundaries that are used in the simulation. Also, the `BlackbodyRadiationManager` class is used to compute the color values that the `ColorLookUpTexture` texture will consist of.

Each of these three classes will be described next.

6.5.2.1 ParticleVisualizer

The `ParticleVisualizer` class implements a particle system that is used to visualize the explosion based on the algorithm described in section 5.4. Instead of handling both fire and smoke particles at the same time, two instances of the class has to be created instead, where one instance handles the fire particles and the other instance handles the smoke particles. The type of the particle system is given as parameter to the constructor along with references to the necessary textures from the simulation step.

Before the simulation starts, the particles are placed randomly in positions close to the location of the initial spark. The maximum distance from the initial spark as well as the number of particles can be adjusted by setting the appropriate parameters.

The movement of the particles, as specified in step 1 in the algorithm, is implemented in the `AdvanceParticles` shader program. This shader program uses the velocity and the turbulence textures to calculate the new position based on equation 5.1. Since both the fire and the smoke particles are moved

based on the same equation, the same shader program is used to move both types of particles.

The rendering of the fire and smoke particles, as specified in step 3 of the algorithm, is implemented in the `RenderFireParticles` and the `RenderSmokeParticles` shader programs respectively. The location and random orientation of the particles are sent to the GPU, and a geometry shader is used to create view-aligned quads oriented based on the particles' random orientations. The color of the fire and smoke particles are calculated based on equations 5.11 and 5.14 respectively, and drawn to the screen using the blending functions 5.13 and 5.16 respectively. These blending functions are implemented by setting the blend states appropriately. Different animated textures are used to render the two types of particles, and the animation frame is chosen based on the elapsed time.

6.5.2.2 ObstacleVisualizer

The `ObstacleVisualizer` class visualizes the internal boundaries used in the simulation step. Boxes are rendered at every obstacle cell, and the boxes are textured using procedurally calculated texture coordinates.

The functionality is implemented in the `RenderObstacleTechnique` shader program. A single vertex is created at each grid position, and the vertices are transformed to fit in the particle domain. The `Obstacles` texture is used to decide whether or not a given cell in the grid is an obstacle cell, and a geometry shader that is called for each vertex is used to render a box at every obstacle cell. Vertices that are not located at obstacle cells are culled. The boxes are given texture coordinates based on the position of the box in the grid. For example, the leftmost box in the x-direction is given a texture coordinate equal to 0, the rightmost box is given a texture coordinate equal to 1, and the boxes in between are given texture coordinates linearly interpolated between these values. The same calculation is done in both the x-, y-, and z-direction.

6.5.2.3 BlackbodyRadiationManager

The `BlackbodyRadiationManager` class calculates a range of 256 color values based on a black-body radiation model. The calculation of these values is described in section 5.3.1. The class consists of two methods that can be used to retrieve these values. The `GetColor` method takes an index as parameter, and returns the color value located at the given position in the range. This index can be any integer between 0 and 255. The `GetColors`

method takes no parameters, and simply returns an array containing all the color values in the range.

6.6 Stand-alone applications

Some of the results that the explosion animation application needs can be computed in advance. Two stand-alone applications have been created to facilitate this: An application to generate Kolmogorov turbulence fields; and an application to create the animations that are used to render the particles.

6.6.1 Kolmogorov turbulence generator

The generation of the 3D Kolmogorov turbulence fields, which are used during particle movement in the visualization step, is quite computationally expensive. However, since the fields do not change during the explosion animation, there is no need to calculate them in the actual explosion application. Thus, a stand-alone application has been implemented to create the turbulence fields.

The equations and algorithm that were described in section 5.2.2 are used to create the Kolmogorov turbulence field. These equations are rather complex and difficult to get a intuitive comprehension of. They are therefore implemented in a straigh forward and naive fashion, making the transition from equations to code as easy as possible.

Most of the difficulties are encapsulated in a class called `KolmogorovFrequencyCalculator`. When instantiated, the constants ϵ, $k_{inertial}$, and σ that were introduced in section 5.2.2 need to be supplied. The class has a single public method, `calculateU`, that creates three complex numbers based on the supplied constants, as well as the temporal position, i, j, k, and l.[2] This corresponds to the calculations of equation 5.8.

The algorithm that is explained in 5.2.2 uses `calculateU` to fill a 3D grid of complex temporal vectors.[3] Next, the grid is transformed into the spatial domain by using an inverse Fourier transform. We use the FFTW library [FJ97] to perform this transition. Finally, the results are saved to a 3D texture, which can be loaded into the explosion animation application using the DirectX API.

[2]The variable l is always set to zero, because we are only interested in a single 3D field each time the application runs.

[3]For instance a 128x128x128 3D grid with 3 complex numbers in each cell.

6.6.2 Animated texture generator

As explained in section 5.3.4, animated textures are used to render the particles. These animated textures are created by blending a static texture splat with each frame in a 3D Kolmogorov velocity field. The resulting animated texture is static and can thus be precalculated. A stand-alone application has been implemented to create such animated textures.

The application is implemented in the `AnimatedTextureGenerator` class, and the class consists of two methods; the `Generate` method, and the `Normalize` method. The `Generate` method takes the path of a static texture splat and a 3D Kolmogorov velocity field as parameters. The textures at the specified paths are loaded and stored in two arrays. Each color value in the static texture splat is multiplied with the velocity value at the corresponding position in the xy-plane in the velocity field. This is repeated for every velocity field slice in the z-direction. The resulting values represents the animated texture values, and they are stored in a 3D texture, which is then returned by the method.

The animated texture that is created by the `Generate` method is not normalized. That is, the various color values does probably not use the entire brightness scale available. In a normalized texture, the brightest color is represented by an all white color instead of a tone of gray. This ensures that a texture is not too dark. For this reason, the `Normalize` method can be used to normalize the resulting animated texture. The method takes an animated texture as parameter, and locates the brightest color value located in the texture. Next, every color value is first divided by this color and then multiplied by the white color value. For example, if the brightest color value in the texture is 192 and the white color value is 255, a color value of 64 would result in a new color value equal to 85.[4]

6.7 Summary

This chapter has presented an overview of our implementation of the simulation and visualization of explosions. The implementation is divided into modules, where each module implements a given set of the necessary functionality. The core module is the GPU computational framework, which provide functionality to make general-purpose programming on the GPU easier. The simulation of the explosion is implemented in it's own module, and makes use of a general fluid solver to calculate the evolution of the velocities and densities that are used in the simulation. Finally, the result of

[4]The new color value is calculated as follows: $\frac{64}{192} \cdot 255 = 85$

the simulation is visualized by the visualization module.

A couple of stand-alone applications have been developed as well. The Kolmogorov turbulence field that is used to add additional turbulence to the explosion is too computationally demanding to be calculated in real-time, but since it can be computed in advance, a stand-alone application has been developed to create such a field. Also, the animated textures that are used to render the particles can also be computed in advance, and a stand-alone application has been developed to create these textures as well.

Chapter 7
Results

This chapter presents the results from the implementation of the method that were proposed in the earlier chapters. The actual discussion of these results may be found in chapter 8. Image captures of the produced explosions are shown, but the videos that accompany this thesis generally illustrate the results a lot better. Note that not all the explosion video clips are shown using their actual frame rate.[1]

7.1 Overview

The simulation and visualization can be evaluated to investigate the effects of the numerous parameters and methods that are used. We have selected a set of evalution topics that we find most interesting to investigate. The chapter is structured as follows:

First, the effect of varying the simulation grid size is tested, in terms of both performance and visual results. Intuitively, increasing the grid sizes should result in improved visual quality at the expense of slower frame rates. We wish to investigate whether this is true. To further investigate what part of the animation algorithm that is the bottleneck, the performance is evaluated with and without performing the visualization step.

Second, various tests are performed at the visualization itself. Since the explosion is visualized using particles, it is interesting to see how the particle count affect both performance and visual quality. Also, as the particles are visualized using screen-aligned quads, it is of interest to consider the effect of varying the quad size. Intuitively, if the number of particles are decreased, their sizes must be increased to provide a sufficient visualization. Also, the effect of the movement factor, which were introduced in chapter 5, is tested. Finally, the effect of using animations to texture the particles is evaluated.

Third, the effect of the vortex particle and the Kolmogorov turbulence methods are evaluated. The vortex particle method is tested by varying the pa-

[1] This is explained further in a readme file in the accompanying media folder.

rameters that are used to configure it. The effect of Kolmogorov turbulence method is shown by adjusting its strength.

Fourth, the effect of the thermal expansion is evaluated. We wish to see what can be achieved by adjusting its strength.

Finally, results from full 3D simulation including internal boundaries are presented, followed by a comparison between the results from our approach and the results of existing methods.

Table 7.1 shows the specifications of the computer that were used when producing the results. All the hardware are available through ordinary computer retail stores, and are likely to resemble equipment of game enthusiasts.

CPU	Pentium 4, 3.0 GHz
RAM	2048 MB
GPU	Gainward GeForce 8800 GTX
GPU Memory	768 MB
GPU Core Clock	575 MHz
GPU Stream Processors	128
Operating system	Microsoft Windows Vista Enterprise
Graphics drivers	NVIDIA ForceWare 158.24

Table 7.1: Hardware specifications and setup.

7.2 The effect of varying the grid sizes

Tests of 2D simulations with volumetric extrusion and full 3D simulation have been performed. This section provides performance and visual results from these tests.

7.2.1 Performance related to grid sizes

As explained in chapter 6, the full 3D simulation considers arbitrary internal boundaries. This involves a large amount of additional code per simulation cell when compared to the 2D simulations. Thus, the results from the 2D and 3D simulations are not directly comparable in terms of performance.

7.2.1.1 Simulation without visualization

Table 7.2 shows the performance results of the 2D simulation, without visualization. The left column shows the various grid sizes that were used and the right column shows frames per second (fps).

Grid size	Simulation only
16x32	767 fps
32x64	771 fps
64x128	768 fps
128x256	592 fps
256x512	184 fps
512x1024	50 fps
1024x2048	13 fps

Table 7.2: Frame rates for 2D slice explosion simulation.

As seen from the table, the simulation performance of the three smallest grid sizes are, in practice, identical. Despite that there is a factor 16 increase of grid cell count from the 16x32 simulation to the 64x128 simulations, the simulation still performs at around 700 frames per second. The performance is obviously not bound by the number of cells at these grid sizes. However, when the grid sizes continue to increase, the simulation performance drops linearly, decreasing at the same rate as the number of grid cells increases. Figure 7.1 illustrates this by using logarithmic scales on both axes.

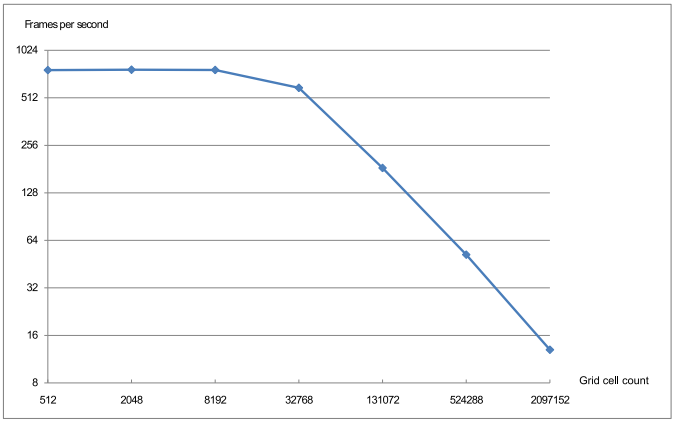

Figure 7.1: 2D Simulation performance without visualization. The vertical axis shows frames per second, while the horizontal axis shows the total amount of simulation grid cells.

Table 7.3 shows the performance of the full 3D simulation. As expected, it is more computationally demanding than the 2D simulation, even for equal cell counts. For instance, the 2D 64x128 and the 3D 16x32x16 has both 8192 simulation grid cells, but the 2D simulations performs at 768 frames per second, whereas the 3D simulation performs at mere 209.

Grid size	Simulation only
16x32x16	209 fps
32x64x32	52 fps
64x128x64	13 fps
128x256x128	1.7 fps

Table 7.3: Frame rates for full 3D explosion simulation without visualization.

As observed for the 2D simulation, one could expect the 3D simulation performance to decrease linearly with the same rate as the number of cells increase. For instance, when increasing the cell count by a factor 8 from 16x32x16 to 32x64x32, a corresponding performance drop of a factor 8 seems logical. However, only a performance drop of a factor 4 is observed, except for the 7.64 performance drop when increasing the grid from 64x128x64 to 128x256x128. Figure 7.2 illustrates how the frame rates decrease as the simulation cells count increases.

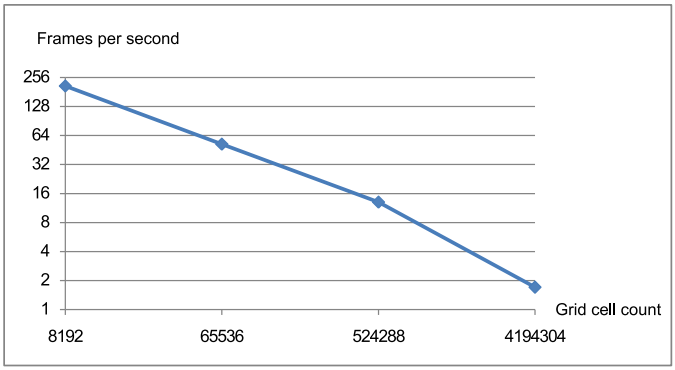

Figure 7.2: 3D Simulation performance without visualization. The vertical axis shows frames per second, while the horizontal axis shows the total amount of simulation grid cells.

7.2.1.2 Simulation and visualization

Table 7.4 shows the performance results of the 2D simulation when including the visualization of 10 000 fire particles and 10 000 smoke particles. The

rightmost column shows the total number of times the pixel shader code for rendering the particles were executed each frame. The number of pixel shader are relevant for the visualization performance because it represent the vast majority of work that is performed by the GPU. All the test cases were rendered from the same camera position and angle to ensure as identical test conditions as possible.

Grid size	Simulation and visualization	Pixel shader invocations
16x32	371 fps	10.8 M
32x64	295 fps	14.4 M
64x128	347 fps	9.7 M
128x256	267 fps	9.2 M
256x512	132 fps	10.8 M
512x1024	38 fps	11.2 M
1024x2048	10 fps	11.2 M

Table 7.4: Frame rates and pixel shader invocations (in millions) for 2D simulations that is visualized by 2x10 000 particles.

One could expect the three smallest grids to perform at equal frame rates as they did without the visualization. However, as table 7.4 shows this is not the case. Especially, the 32x64 simulation performed at lower frame rates than both 16x32 and 64x128. Also, we can observe that the number of pixel shader invocations for the 32x64 simulation is well above the average.

Grid size	Simulation and visualization	Pixel shader invocations
16x32x16	111 fps	19.7 M
32x64x32	32 fps	14.2 M
64x128x64	9 fps	16.6 M
128x256x128	1.4 fps	16.4 M

Table 7.5: Frame rates and pixel shader invocation (in millions) for the full 3D explosion simulation, visualized by 2x10 000 particles.

Table 7.5 shows the frame rates and number of pixel shader invocations per frame of the full 3D simulation when including the visualization. Is is obvious that the 2D explosion outperforms the 3D explosion, but again, the implementations are not directly comparable in terms of performance.

To further investigate the performance impact of the visualization step, the 2D explosions have been rendered at different distances. Figure 7.3 shows three screen captures from a test case. The leftmost screen capture shows a 2D simulation rendered with a camera far from the explosion, which per-

formed at 373 frames per second. The rightmost screen capture is taken while rendering the explosion up close, which performed at 49 frames per second. The middle screen capture performs at 239 frames per second. Clearly, the performance drops dramatically when close-ups of the explosions are rendered, which is also observed by the number of pixel shader invocations. The pixel shader invocations for the far, middle, and close-up rendering were 5.0, 10.1, and 87.9 millions, respectively.

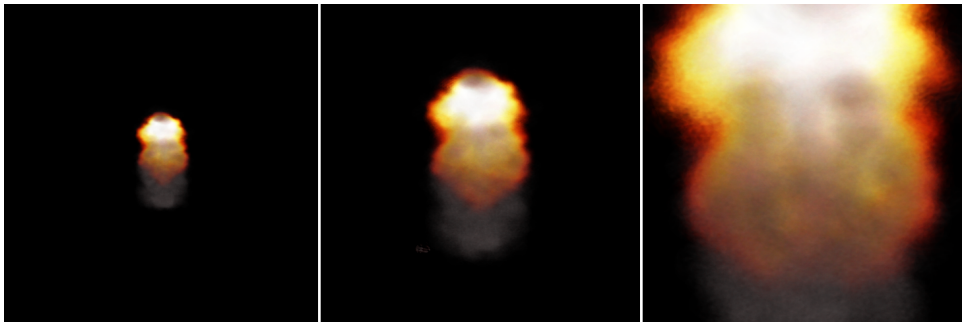

Figure 7.3: A 2D simulation at three distances. Far view (left) @ 373 fps, middle view @ 239 fps, near view @49 fps. (2x10 000 particles)

7.2.2 Visual results

The visual results that is produced vary based on the grid size. The various parameters that control the simulation must be adjusted from one grid size to another to produce desirable results, but it is difficult to achieve similar results in all the grid sizes.

7.2.2.1 2D Simulation

Figure 7.4 shows four selected screen captures from two 2D simulations. The top four pictures are from the smallest grid size of 16x32, while the bottom four are from a 32x64 grid. Both have rather dark orange colors and large amounts of smoke early in the simulation, representing dense exhaust gas with low temperatures. The 32x64 explosion reaches higher intensity fire color than its counterpart, but neither seem able to reach high temperatures. Surely, the simulation parameters may be adjusted to generate high temperatures, but this results in explosions that are not able to calm down before occupying the entire simulation domain. The motion of the explosions is fast and chaotic, but with limited rolling characteristics. We experience that simulations at these grid sizes change significantly, even when very

small adjustments of the simulation parameters are made. This making the simulations difficult to customize.

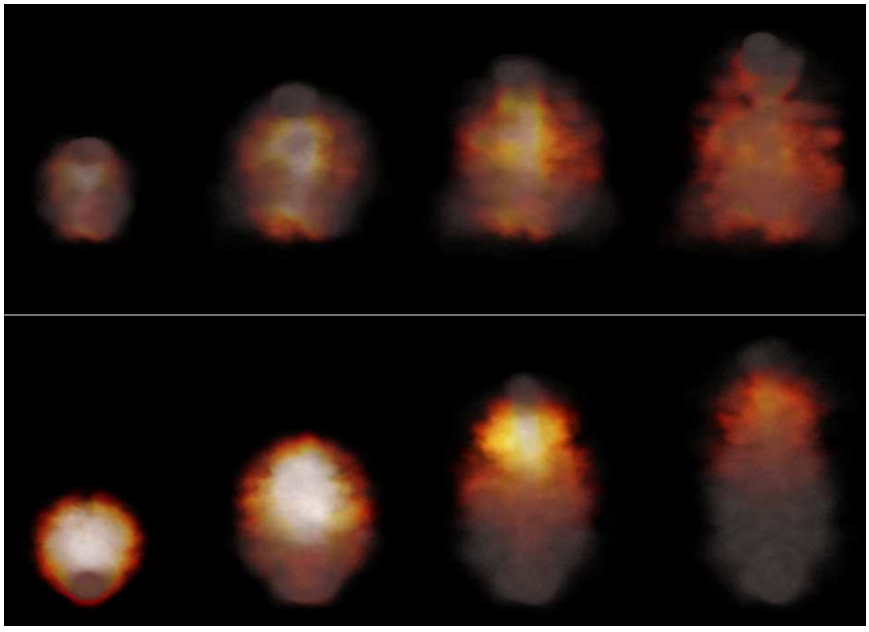

Figure 7.4: Four selected screen captures of the evolution of 2D simulations on a 16x32 grid (top) and a 32x64 grid (bottom).

Medium sized grids, such as 64x128, 128x256, and 256x512 are less chaotic and easier to control than their smaller grid counterparts. The motion is less turbulent, but the shape and overall motion of the explosions are more believable. Compared to the smaller grids, the temperature reaches higher levels before cooling down, as indicated by the brighter colors. Figure 7.5 shows an example of an explosion simulated in a 128x256 grid.

The largest grid sizes that have been tested are 512x1024 and 1024x2048. Simulations produced using these grid sizes are slow, have small amounts of turbulence, and do not have the rolling motion of explosions. In fact, the results resemble laminar buoyant plumes more than highly turbulent explosions. As with the medium sized grids, the colors show that the temperature reaches high levels before cooling down. Figure 7.6 shows screen captures from explosions simulated in large grids. The top four images are from a 512x1024 simulation, while the bottom four is from a 1024x2048 simulation. As seen from the pictures, the results from the 1024x2048 simulation seem more laminar that the results from the 512x1024 simulation.

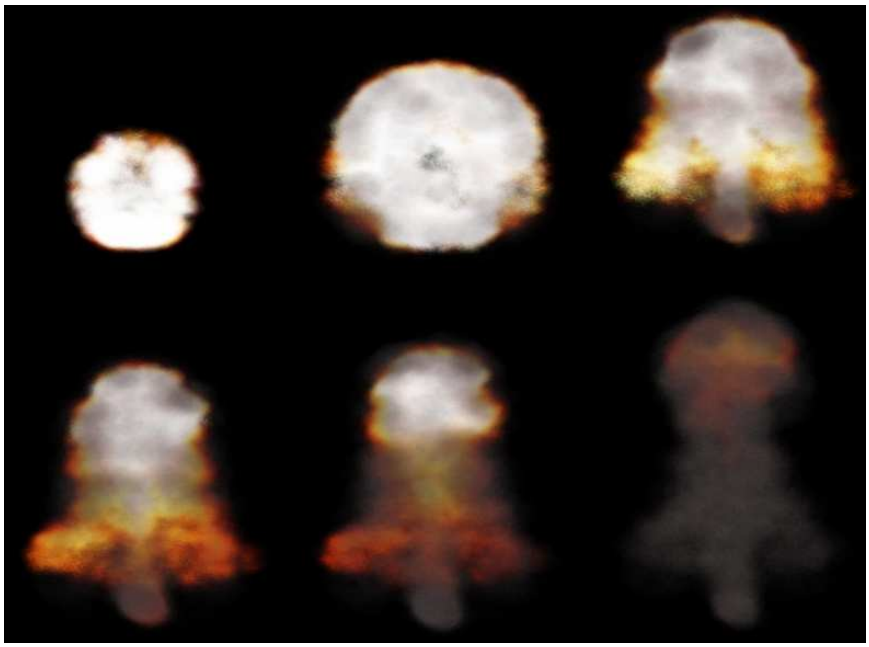

Figure 7.5: Six selected screen captures of the evolution of a simulation on a 128x256 grid.

7.2.2.2 3D Simulation

Due to the computational requirements of full 3D simulation, we were not able to perform tests with such a variety of grid resolutions as with the 2D simulations, thus only three different grid sizes are discussed. In our experience, achieving desirable results from the 3D simulation is more difficult than for the 2D simulations, thus requires more time to adjust the simulation parameters.

The smallest grid that were tested had the dimensions 16x32x16. Results using this grid size is shown by figure 7.7. Simulations at this grid size have similar problems as the small grid 2D simulation, but they are not that severe: The 3D simulations performed at small grids are difficult to adjust and small changes of simulation parameters result in significant changes. Also, explosions at small 3D grids also have problems reaching high temperatures before they should calm down. Their motion is turbulent, but has limited rolling characteristics.

As for 2D explosions, 3D simulations in medium sized grids are easier to control than the smallest grids. A medium grid of size 32x64x32 are used to create the results shown in figure 7.8. The simulation produces turbulent

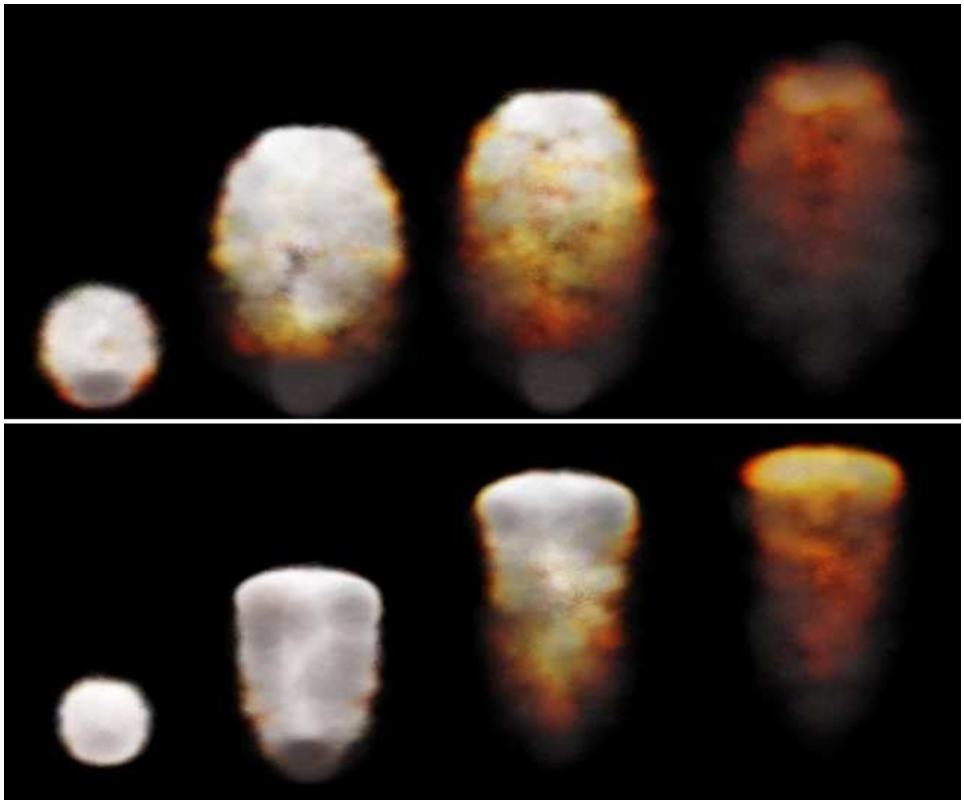

Figure 7.6: Selected screen captures of 2D simulations on a 512x1024 grid (top) and a 1024x2048 grid (bottom).

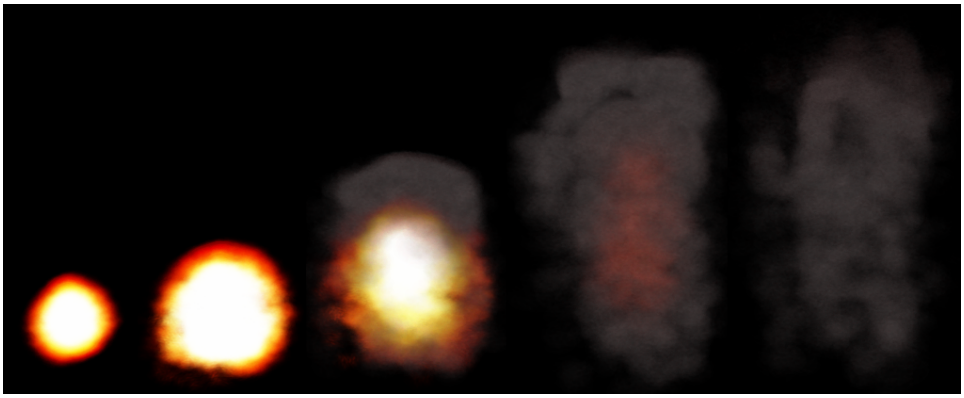

Figure 7.7: Screen captures from a 3D simulation using a 16x32x16 grid.

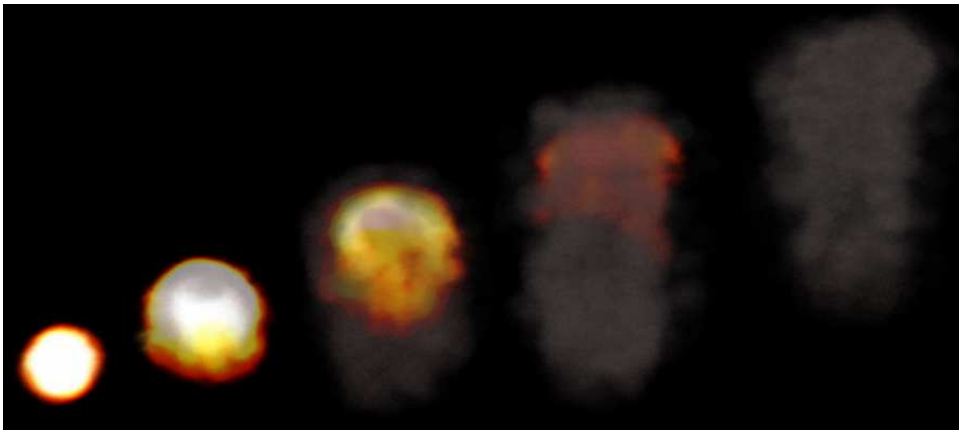

Figure 7.8: Screen captures from a 3D simulation using a 32x64x32 grid.

motion with appealing rolling motion. The rapid expansion that should occur at the beginning of the simulation is not very eminent, however. Instead, the highest velocities during the simulation is after a few seconds when the fire ball moves upwards due to buoyancy. [2]

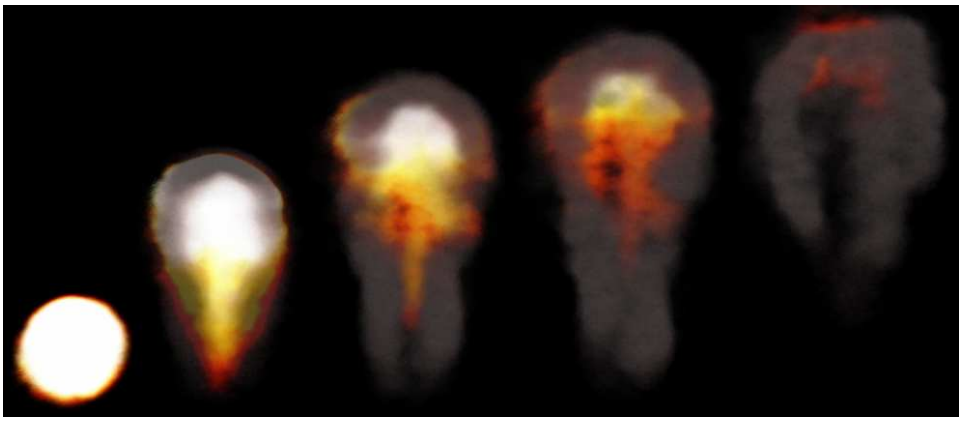

Figure 7.9: Screen captures from a 3D simulation using a 64x128x64 grid.

The largest grid that were used to simulate explosions in full 3D was 64x128x64. Examples of the produced results are shown in figure 7.9. The simulation looks laminar at first, but appealing turbulence and rolling motion can be observed after a short while. As presented in section 7.2.1.2, simulations using 64x128x64 grids perform at around 10 frames per second. Combined with the fact that the 3D explosions are rather difficult to adjust, produc-

[2] Can be observed in the video file called 3D_GridSize.wmv

ing great visual results may be cumbersome. For instance, the shape of the smoke in the end of the explosion shown in figure 7.9 is not very believable. By spending more time adjusting the simulation parameters, we suspect that better results than the ones shown in figure 7.9 can be produced.

7.2.2.3 2D simulation compared to 3D

Because the 2D simulation are more computationally efficient than their 3D counterparts, it is interesting to evaluate whether 2D simulations can be used to create sufficient visual results. Since the 3D simulations are the only ones able to include internal boundaries, the 2D simulations are only a valid option when there are no interfering obstacles near the explosions.

Figure 7.10 shows a 2D explosion simulated using a 64x128 grid side-by-side with a 3D explosion simulated using a 64x128x64 grid. Generally, we think they are rather similar. There are some visual differences that is caused by varying temperature and exhaust gas levels, but such differences can also be seen when comparing 2D explosions with unequal simulation parameters. However, the results from 2D and 3D explosions differ in two main ways: The 3D simulations generate more complex and rolling motion, while the 2D explosions are generally easier to adjust by altering the simulation parameters.

7.3 Effects of varying particle properties

The particle systems that are used to visualize the explosion have several properties that can be adjusted. The following sections test the effect of varying them, using a 2D simulation with grid size 64x128.

7.3.1 Varying particle count and sizes

Section 7.2.1.2 showed the frame rates' dependency on the number of pixel shader invocations that is used to renderer the particles. Thus, it is of interest to look at the effect of varying both the particle count and their sizes. Clearly, a certain amount of particles are needed to avoid being able to see through the explosions. Thus, if the number of particle are reduced, their sizes may have to be enlarged to ensure visual quality.

To investigate further, various particle counts have been used. First, frames per seconds were recorded by the various particle count, without adjusting their sizes. Next, frames per seconds were recorded after adjusting the par-

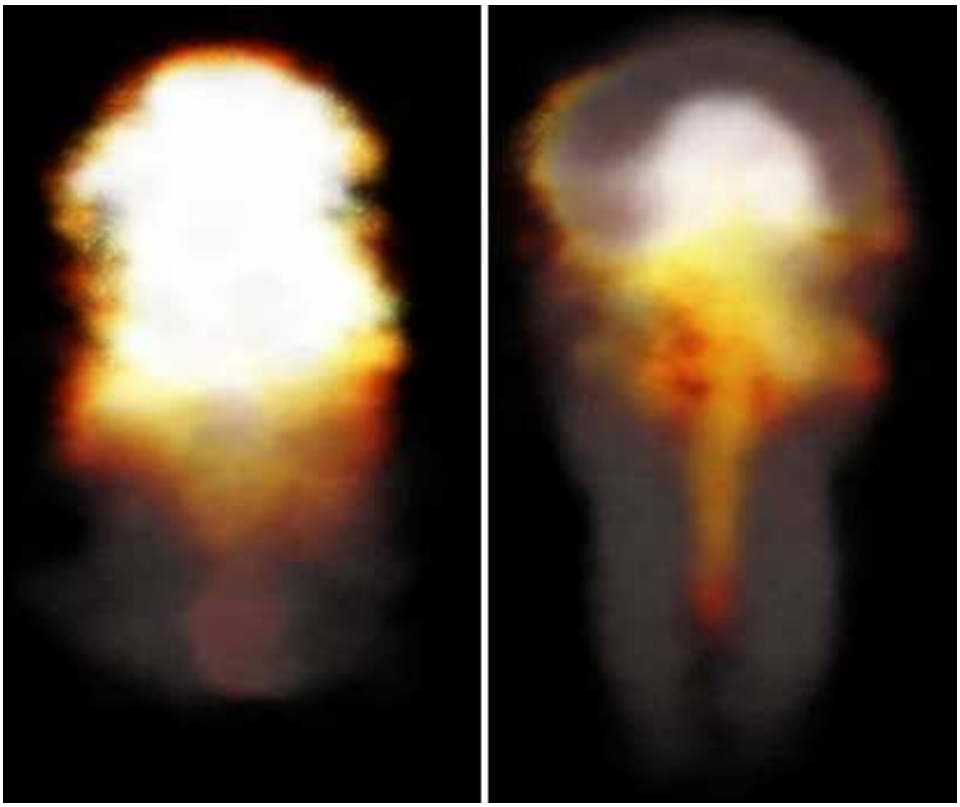

Figure 7.10: Screen captures of simulations simulated in 2D (left) and 3D (right).

ticle sizes to maximize the visual quality. A visualization using 2 x 10 000 particles (10 000 fire and 10 000 smoke particles) is used as a basis for comparison, as such a configuration seems to work well in most cases. The base case runs at 316 frames per second.

Particle count	**Fps without size adjustment**	**Fps with size adjustment**
2 x 2 500	371 fps	331 fps
2 x 5 000	353 fps	328 fps
2 x 20 000	223 fps	297 fps

Table 7.6: Frames rates for various particle counts, both before and after adjusting their size for visual quality.

The frame rates recorded from using various particle counts are summarized in table 7.6. As expected, the frame rates increase when reducing the particle counts from 2 x 10 000 to 2 x 5 000 or 2 x 2 500, and decreases when using 2 x 20 000. However, when the particle sizes are adjusted the differences are not that obvious. Note that the particle sizes are *decreased* for the 2 x 20 000 case, while *increased* for the two other test cases.

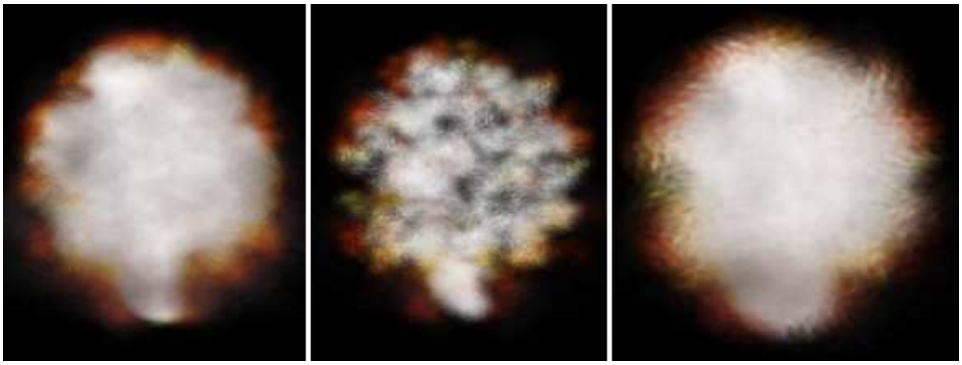

Figure 7.11: Shows an explosions visualized using 2 x 10 000 particles (left), 2 x 2 500 particles without size adjustment, and 2 x 2 500 particles with size adjustment.

Figure 7.11 illustrates the actions that were taken to test the effect of adjusting the number of particles and their sizes. The leftmost picture is the base case of 2 x 10 000 particles, the middle shows the same simulation step, but visualized using 2 x 2 500 particles without size adjustment. The rightmost picture shows the result after adjust the particle sizes of the middle picture. The middle picture clearly illustrates that the visualization suffers from reducing the particles without adjust their sizes also. The rightmost

picture shows that reasonable results can be produced by increasing their size, but the leftmost picture of 2 x 10 000 particles has in our opinion the best visual quality.

7.3.2 The effect of the movement factor

Chapter 5 introduced a *movement factor* that is used to create variety in how and where the particles move. To illustrate the movement factor's effect, a single simulation has been visualized by two differently configured particle systems. First, the movement factors are chosen randomly from a defined distribution. Second, the movement factors are uniformly set to 1.0, which represent how the particles would move without using the movement factor method. The left and right pictures of figure 7.12 shows the respective results from the visualizations using randomly chosen and uniformly chosen movement factors. Clearly, the particles of the leftmost picture visualize the explosion in a more suitable way than the particles of the rightmost picture. No performance loss has been observed by utilizing the movement factor method.

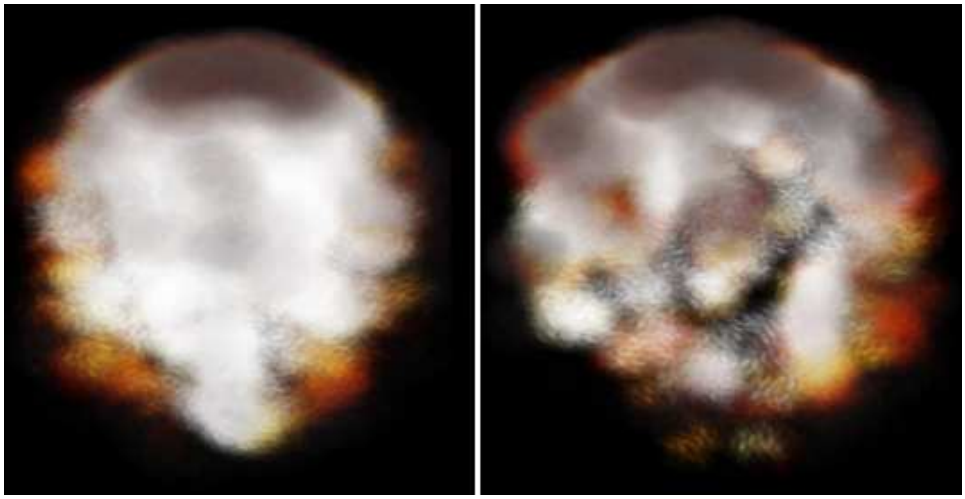

Figure 7.12: A simulation visualized using varying movement factors (left) and uniform movement factors of 1.0 (right).

7.3.3 The effect of animating the particles

Chapter 5 also proposed to texture the particles with animations created from Kolmogorov turbulence fields. Contrary to our expectations, this does

not seem to have a noticeable effect on the final results. Only if the camera is very close to stationary smoke, we can observe a small increase of small-scale movement.

However, the use of animations results in a small decrease in frame rates. Tests performed using a 2D grid of 128x256 performed at 208 frames per second when using animation, while performing at 214 otherwise.

7.4 Turbulence modelling results

Two methods for introducing turbulence to the explosions were proposed in chapters 4 and 5. The following two sections present their effect on the final results.

7.4.1 Vortex particles

The vortex particle method is used to introduce rotational motion to the velocity field. Several tests have been performed to illustrate the effects of varying the number of vortex particles, their radii, and strengths. One parameter at a time is raised to a high, but reasonable level, while the others are kept rather low. Table 7.7 shows how the parameters were adjusted during the tests. The results are shown in figure 7.13.

Picture position	Particle count	Max radius	Max strength
Left	4000	4 grid cells	1x
Middle	400	6 grid cells	1x
Right	400	4 grid cells	2.5x

Table 7.7: Parameter values for the vortex particle method used to create the pictures of figure 7.13.

The motion that are generated by the vortex particle method is turbulent, but creates rather strange and un-natural shaped explosions, as can be seen from figure 7.13. Furthermore, the particle parameters that were adjusted do not seem to have their own distinct effect on the result. The pictures in the figure do not differ enough to be able to couple their appearance to the parameters. No performance loss has been observed by the use of vortex particles.

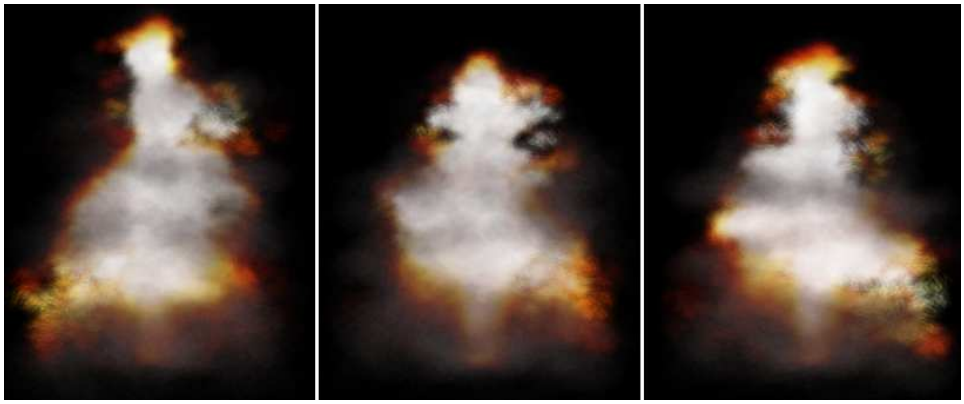

Figure 7.13: Simulations using the vortex particle method with different parameters. From left to right: Large particle count, large radii, and powerful strengths.

7.4.2 Kolmogorov turbulence

The second method that were implemented to include turbulence was the Kolmogorov turbulence method. A turbulence velocity field was used in addition to the velocity field from the simulation to move the particles of the visualization step. The results that were created with Kolmogorov turbulence were far more predictable and controllable than those created with the vortex particle method.

Figure 7.14 illustrates the effect of using Kolmogorov turbulence when visualizing an explosion. The leftmost picture shows the explosion visualized without Kolmogorov turbulence, while the middle and rightmost figures are visualized using medium and high levels of Kolmogorov turbulence, respectively. The largest differences between the picture created without turbulence and the two others in figure 7.14 is the levels of smoke. However, some increase in turbulence can be observed. Note that the effect of Kolmogorov turbulence is far more visible in motion than in the still pictures.

No performance loss were observed using the Kolmogorov turbulence method.

7.5 Internal boundaries

The full 3D simulation that has been implemented has the option of including internal boundaries. To demonstrate what can be achieved with these kinds of simulations, three examples are given. All three examples were

Figure 7.14: An explosions visualized with no (left), medium (middle), and high (right) levels of Kolmogorov turbulence.

rendered with 2 x 20 000 particles at 31 frames per second.

Figure 7.15 shows an explosions that occurs inside an immobile arch. The geometry of the arch causes the explosion to burst outwards in the horizontal directions before rising due to heat. Figure 7.16 shows an explosion near the ground, next to a wall. Note how the explosion is blocked by the wall. Figure 7.17 shows an explosion that occurs below a flat horizontal obstacle. The explosion is forced to move around the obstacle, which causes interesting motion of the fluid.

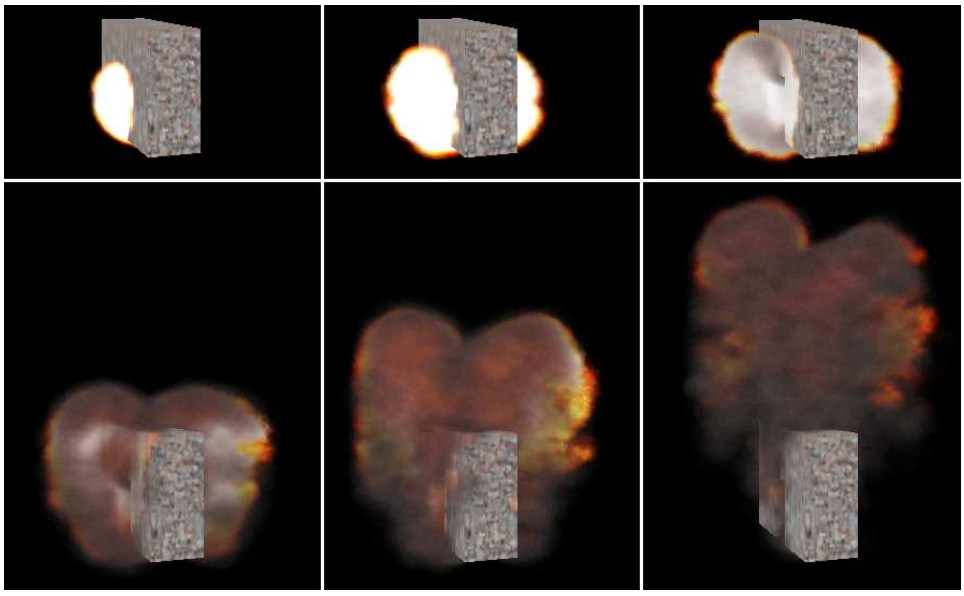

Figure 7.15: Six frames showing the evolution of full 3D simulation of an explosion inside an immobile arch (31 fps).

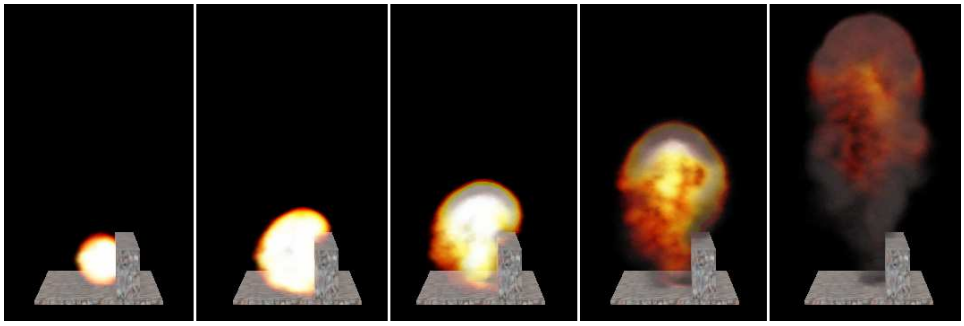

Figure 7.16: Five frames showing the evolution of a full 3D simulation of an explosion near a wall (31 fps).

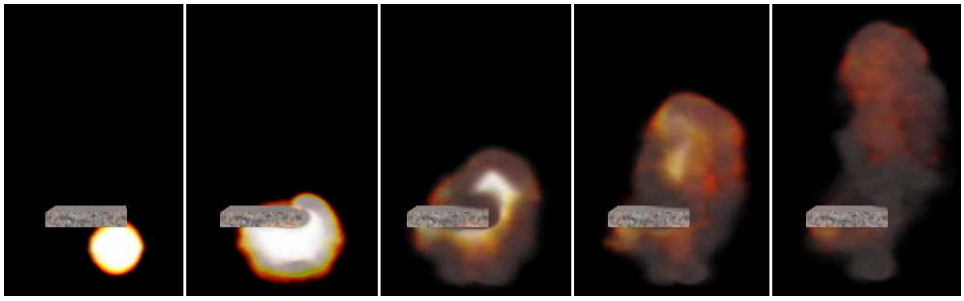

Figure 7.17: Five frames showing a full 3D simulation of an explosion under a flat obstacle (31 fps).

7.6 The effect of thermal expansions

To model the thermal expansions that occurs during combustion is central to the simulation. Tests performed with various amounts of expansion is shown in figure 7.18. Each horizontal pair of pictures in the figure represent two different time steps from a simulation. The topmost pair shows an explosion without the use of thermal expansion. Note how small the explosion is. The middle pair shows an explosion using a medium amount of expansion, resulting in a significantly larger explosion. The bottommost pair of pictures shows a simulation using a large amount of expansion. The pictures show that this explosion expands even further. Also interesting is the small, dark area within the final explosion. It indicates that there too few particles present in the center of the explosion to visualize it properly.

When performing 3D simulations with internal boundaries, the thermal expansion is used extensively to force the fluid around the boundaries near the point of impact (i.e. figure 7.15). An important observation made through adjusting such simulations is that the thermal expansion can only be used to a certain degree. When the thermal expansion strength reaches a certain value, increasing it further does not seem to generate more powerful explosions. Instead, some of the momentum of the fluid seems to disappear in an unnatural fashion. Increasing the accuracy (number of jacobi iterations) of the projection step does not seem to solve this issue.

Also, interesting is how the thermal expansion affect the density fields. During the first few moments, the thermal expansion seems to increase the amount of density cells with large values. Is it quite obvious that the total sum of density values when using thermal expansion is far greater than it is when the combustion model operates on its own.

7.7 Comparison with other approaches

This section contains a comparison, in term of visual quality and performance, between our approach and other methods for animating physically based explosions. [KW05] is the only such method we know of that performs in real-time. Therefore, a few offline approaches are included in the comparison as well. Still images and video captures[3] are used to compare

[3]Videos for the various approaches can be found at the following locations (as of 20. June 2007):

- [FOA03]: http://www.cs.berkeley.edu/b-cam/Papers/Feldman-2003-ASP/
- [KW05]: http://wwwcg.in.tum.de/Research/Publications/VolEffects/

We were not able to find video footage of the explosion from [RNGF03].

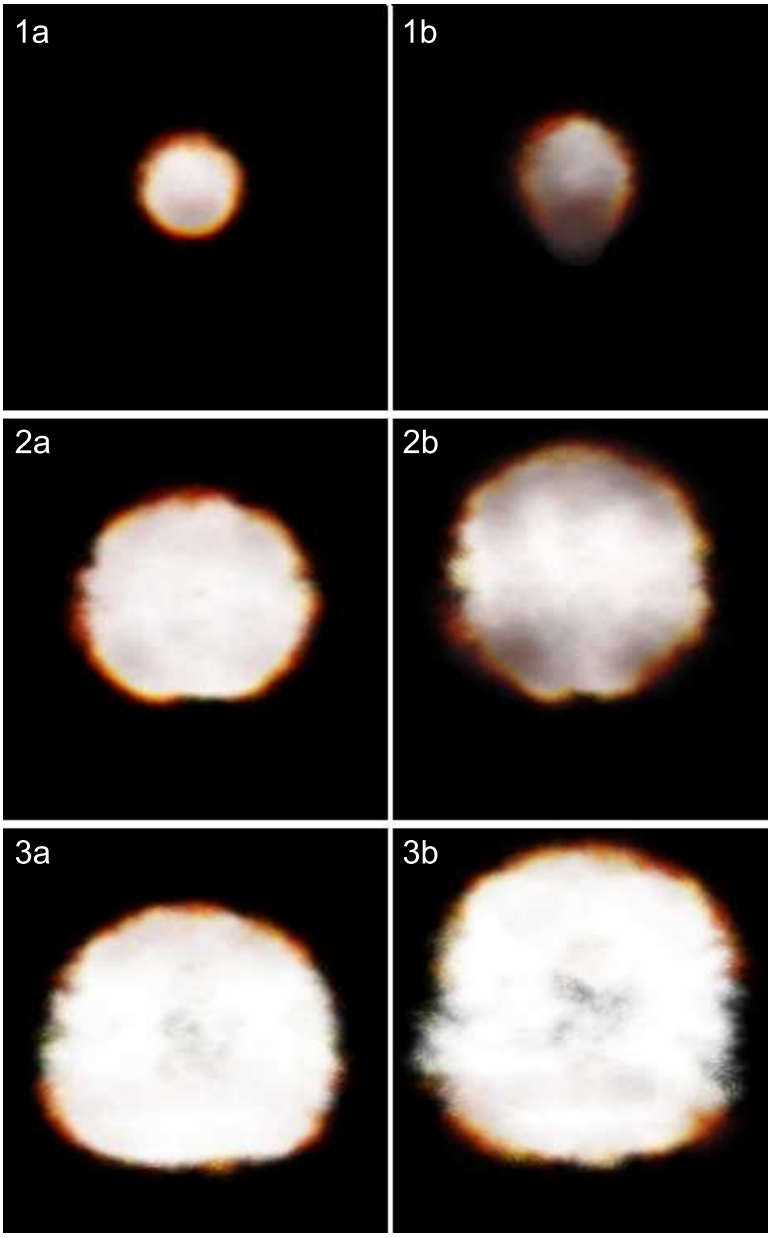

Figure 7.18: Two time steps from three explosions simulated using none (1a, 1b), medium (2a, 2b), and large amounts (3a, 3b) of thermal expansion.

the visual results. To guide the visual comparison, the following criteria is used: color and texture, and shape and motion.

The colors of the explosions created by [KW05] are visually pleasing. However, do not seem to visualize a dynamic range of temperature as well as the resulats from our method, because their colors are mostly orange or red. Our colors, on the other hand, are somewhat over-saturated in some of the results, and appear less colorful. Different from their method, our method models smoke that can even remain after the fire ball has disappeared. We have not seen results from [KW05] that show a complete explosion from start to end, so we do not know how their explosions look like when calming down. The texture of our explosions seem slightly more detailed than of the ones of [KW05]. Contrary to their results, we experience some visual artifacts, especially near the edges of the explosions, that is caused by particles being rendered alone[4]. The left picture of figure 7.19 shows show a result of our approach, the middle picture is from [KW05], and the rightmost picture is from [RNGF03].

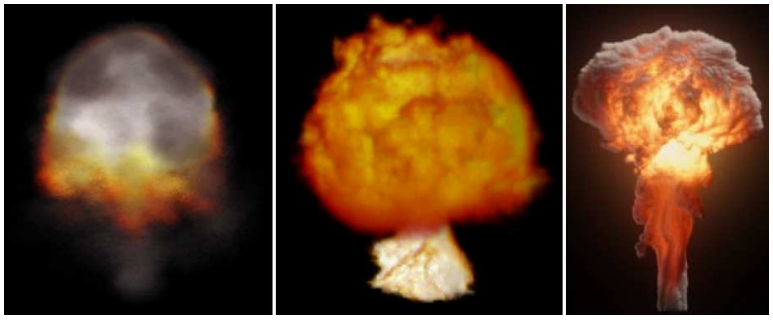

Figure 7.19: Visual comparison of our approach (left), [KW05] (middle), and [RNGF03] (right).

The colors and texture of our method fall short when compared to offline methods such as [RNGF03] and [FOA03], since they use more detailed and time consuming methods for visualization. Especially the results from [RNGF03] are very convincing compared to ours, mainly because of the great detail of the smoke and realistic brightness caused by heat.

The motion and shape of our explosions compare favorable to the one of [KW05], especially during the first few moments. Our explosion has a rapid expansion followed by a rising fire ball, whereas the explosion created by [KW05] moves rather slowly during the entire simulation[5].

The motion and shape of our explosion are not realistic when compared

[4]See for example the top view footage from the 3D Arch video that is attached

[5]This becomes very clear when viewing their video footage

to [FOA03] or [RNGF03]. Especially the video footage of explosions from [FOA03] shows more realistic rolling motion, which is also further enhanced by internal boundaries. Figure 7.20 shows different time steps from a simulation performed by [FOA03] that has similarities to our results in figure 7.15. It is clear from the pictures that our results are inferior to those of [FOA03].

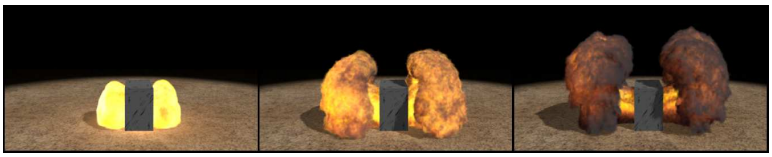

Figure 7.20: Visual results from [FOA03] with similar internal boundaries as our results presented in figure 7.15.

[KW05] used two 2D simulation combined with volumetric extrusion to create explosions. When performing these simulations on 128x128 grids and rendering the results with 16384 particles, they achieved a frame rate of 190. Our simulation of 128x256 (whose cell count correspond to two 128x128 simulation) performs at 278 frames per second when using the same amount of particles. However, [KW05] used a NVidia GeForce 6800 GT, which is inferior to the graphics hardware we used. Comparing performance with the two offline methods is not of interest.

7.8 Summary

We have presented both visual results from our approach for animating explosions. The observations that have been made can be summarized as follows. They are numbered to enable us to refer to them from the discussion in chapter 8.

Res1: The three smallest grid sizes of the 2D simulation, with no visualization, practically performed at equal frame rates (see table 7.2).

Res2: The visualization step is generally the bottleneck of the explosion animation. Tests show that the visualization performance coheres with the number of pixel shader programs that need to be invoked.

Res3: The frame rates of the 3D simulation, with no visualization, did not decrease as much as we expected when increasing the grid sizes (see table 7.3).

Res4: The most visually pleasing explosions are obtained by simulations in medium sized grids. In our experience, the simulation in these grids are also easier to adjust than in grids of both smaller and larger sizes.

Res5: It is easier to create pleasing visual results by adjust the simulation parameters for the 2D simulation than for the 3D simulation.

Res6: A comparison between 2D slice simulations and the far more computationally expensive full 3D simulation show that the 2D simulations may suffice for explosions that do not need to be affected by internal boundaries.

Res7: Reducing the number of particles in the visualization step can improve performance. However, when adjusting their sizes to avoid loosing visual quality we observe that no substantial performance gains occur.

Res8: The use of the movement factor is demonstrated to improve the particle system's ability to render the explosion.

Res9: The use of animated textures on the particles were found to have little or no effect on the final results.

Res10: The vortex particle method produces turbulent motion in the velocity, but the results are rather strange. The use of Kolmogorov turbulence create far more predictable results.

Res11: The thermal expansion is shown to be very important for the simulation, but its strength can only be raised to a certain degree.

Res12: The thermal expansion increases the total amount of density.

Chapter 8
Evaluation

This chapter contain a discussion of the performance and visual results that were presented in chapter 7. Next, an evaluation of the results is performed based on the requirements that were set in section 1.2. Then, our main contributions are summarized. Finally, the thesis is concluded and suggestions for further work are proposed.

8.1 Discussion

The previous chapter presented both solid test results and observations that were made by ourselves while working with the explosion animation. This section contains a discussion of the results that we find most interesting. The visual results are discussed in light of how easy they are to achieve, as well as brief discussion of when the 2D or 3D simulation could be put to use, respectively. Also, the performance results are discussed, because they were not quite as we had expected. Furthermore, we discuss the effect of the vortex particles and Kolmogorov turbulence. A few points are made regarding the visualization step, before discussing some of issues that were observed with the thermal expansion.

8.1.1 General visual quality

When working with physically based explosions, the values of several parameters, such as combustion and force strength parameters, need to be chosen. Adjusting these can be a very cumbersome task. Achieving good visual results with both the smallest and largest 2D grids are difficult. Simulations using the smallest grids are very sudden and hard to control. It is difficult to adjust the simulation parameters such that the explosion reaches high temperatures and violent motion and then calm down again. The simulation domain is probably too small. Achieving violent and turbulent motion when using the largest simulation domains are also difficult. The motion is slow and almost laminar. The best results are created using medium sized

grids, where the simulation produces violent and turbulent motion can also be controlled (**Res4**).

Also, in our experience, the 2D simulation are far easier to work with than the ones in 3D (**Res5**). A contributing factor to this may be that the 3D simulations obviously have more degrees of freedom than the 2D simulations. But also, the 3D grid that we used to most were 32x64x32 and is actually rather small. We had problems adjusting the parameters for the corresponding 2D grid of 32x64 as well.

Despite problems with adjusting parameters, the 3D simulation can support a much wider range of effects. Internal boundaries, wind, simulation domain advection [SR06] can, among others, be included. It is not trivial how these could be well incorporated in a 2D simulation when using cylindrical volumetric extrusion. On the other hand, such effects are not always necessary. The results showed that 2D simulations provide good visual results for cylindrical symmetric explosions and at much better frame rates than full 3D simulation (**Res6**). Thus, they may often be a good alternative to full 3D simulations.

8.1.2 General performance

The results (**Res1**) show that the simulations using the three smallest 2D grids perform at practically the same frame rates when not being visualized. Intuitively, we would assume that simulations using 16x32 would be faster than simulations using the 64x128, which has a factor 16 more cells. It is difficult to draw any conclusions of why these results occur, because there are so many different possibly bottlenecks at the GPU. Nevertheless, we suspect the performance to be bound by the number of times the GPU needs to empty its processor pipelines. We have not investigated this any further.

When including the visualization, we saw that it was more time consuming than the simulation (**Res2**). This became especially clear when viewing the explosion up close. We also saw differences in performance results when using the three smallest grids. The performance corresponded somewhat to the number of pixel shader invocations that were performed by the visualization, but this was not the case when comparing the performances of the 16x32 grid and th 64x128 grid. Again, it is difficult to conclude anything without further investigations, but we suspect the visualization of small simulation result grids to perform better due to cache locality.

Furthermore, we expected the performance of the 3D simulation (without visualization) to decrease by a factor 8 when increasing the total cell count

by 8 (**Res3**). Surprisingly, the performance drop was not that significant, except between the two largest 3D grids. Once more, conclusions cannot be drawn without further investigations. However, the fluid solver shader code that is performed per pixel in the 3D simulation fields reads more values from its surrounding than the corresponding 2D code and there are performed more texture fetches at the same locations in total. Thus, we suspect that the lower performance loss than expected may be caused by good cache locality. The fact that there is a factor 8 performance loss between the two largest 3D simulation grids suggest that this is bound by a different, and currently unknown, bottleneck than the smaller 3D grids.

8.1.3 Turbulence

Two methods for including turbulence to the explosion animation have been evaluated. Vortex particles are used directly in the simulation, whereas Kolmogorov turbulence is used to displace the particles of the visualization step. The use of Kolmogorov turbulence was successful in which id introduced predictable, small-scale turbulence. The results when using vortex particles were not that positive.

As **Res10** states, explosions created using vortex particles look unnatural. The underlying velocity field that is used to move the vortex particles has very high values, especially during the first few simulation steps. A possible explanation of why the method creates unnatural results when used in our method may be that they are moved too far by these velocities. Thus, the particles may introduce rotational motion to areas where such motion is unlikely. Also, the fact that the method is used with the 2D simulation, and not 3D, may contribute to the strange results when the 2D slice is extruded to 3D.

Furthermore, when implemented in 2D, the vortex stretching term[1] that alter the vortices is omitted, thus some of the interesting effects that evolving rotational motion could introduce is not included. We expect the method to be more suitable if used in the full 3D simulation.

8.1.4 Particle visualization

The explosions are visualized using particle system. The use of particles have certain advantages, such as texturing that imitates small-scale variations in the fluid and their ability to be affected by visualization step turbulence (Kolmogorov turbulence). However, both **Res11** and **Res8** show that dis-

[1] See section 4.2.3.3 for an explanation of the vortex stretching term.

advantages with using particle systems as well. First, since the particles are moved high velocities, quite a few of them will move to areas outside the explosion and will not be visible. Nevertheless, valuable time is still spent rendering them. Second, one is not guaranteed that the entire simulation is in fact visualized. There might be areas of interest where no particles are positioned. We introduced a movement factor to improve the particles' ability to cover such areas, but a large number of particles is still required to produce good visual quality.

To increase the small-scale variations of the visualization, we proposed using Kolmogorov turbulence to create animated textures to use when rendering the particles. Despite that animated textures have been shown effective by others [Mic06], the results (**Res9**) show that there were little or no effect of the animations, compared to using non-animated textures. We suspect the animation to have such limited effect because the explosions are visualized by thousands of particles being blended on top of each other. In comparison [Mic06] only used a few hundred. We believe that the method we proposed for generation of these animations can prove useful in other contexts, where fewer, and perhaps larger particles are blended together.

Tests were run to find out whether the number of particles in the visualization step could be reduced to improve performance. The results showed that this was only partially true (**Res7**). When decreasing the particle count, their sizes must be increased not to loose too much visual quality. Increasing particle sizes leads to additional pixel shader invocations, which is most likely the bottleneck of the visualization. In total, the performance gains were small.

8.1.5 Thermal expansion

An interesting observation (**Res11**) were made regarding limitations to how thermal expansion is used in our method. Increasing the strength of the expansion is only useful to a certain degree. We believe this behaviour has to do with the velocity advection step of the fluid solver. As explained in earlier chapters, the advection step uses the current velocity to trace backwards to find new values. Then, if the velocities of the expanding circle (sphere in 3D) are high enough, the advection would trace beyond the centre of the explosion to find new velocity values. These new velocities would then be pointing in the opposite direction of what is correct, and perhaps counteract the outward-going motion.

Also worth noticing is that the use of thermal expansion has a tendency to introduce more density to the simulation (**Res12**). The density advection step is the most probable cause, because when using velocities to trace back

to find the new values, there in no guarantee that multiple cells get their new values from the same dense cells.

8.2 Requirement evaluation

This section contains a discussion to what degree we have fulfilled the requirements that were listed in chapter 1.

R1: A physically based simulation of explosions should be performed.

In order to perform a physically based simulation of explosions we have developed the model that were proposed in chapter 4. A combustion process is used to model burning fuel, which produces exhaust products and heat. Buoyancy and gravity is modeled to affect the motion of the fluid in which the explosions occur. The motion of this fluid is governed by the Navier Stokes equations for incompressible flow, while the violent expansion of explosions are modeled using the thermal expansion method. We do not model the shock wave of explosions or the explosions' affect on its surroundings. Even though certain simplifications of the underlying physics are made to meet the real-time requirements, we consider requirement **R1** to be fulfilled.

R2: The overall motion of the explosion should be convincing.

By using the thermal expansion method, the simulation creates a rapidly expanding fire ball during the first few moments. Subsequently, the burning gas rises due to buoyancy. Some of the desired rolling motion are observed. However, we are not able to exaggerate neither the thermal expansion or the rolling motion as much as we would like. Also, the motion can not be mistaken as a real explosion. Thus, we consider requirement **R2** to be partially fulfilled.

R3: It should be possible to place the explosion in a virtual scene.

We have not focused on creating realistic scenes where explosions are likely to happen. Instead, we have visualized the boundaries that affect the explosions and placed the explosions correspondingly, showing that they may

be placed within a virtual scene. However, the explosion do not affect their surroundings. Thus, we consider requirement **R3** to be partially fulfilled.

R4: The simulation should be affected by obstacles within the scene.

As shown by the figures in section 7.5, obstacles can be placed within a scene and affect how the explosions behave. The results are pleasing, but the simulations are rather difficult to configure using the explosion parameters. We consider requirement R4 to be fulfilled.

R5: The visualization should be convincing, thus have realistic shape, color, and turbulence.

The shapes of the produced explosions are closely related to the motion that were required in **R2**, thus the shapes suffer from some of the same limitations that the motion does. We think the shape of the explosions are visually pleasing, but similarly to the motion, we miss the ability of creating "mushroom-shaped" explosion, where the rolling motion is exaggerated.

The colors of hot exhaust gas is approximated by the black-body radiation model, while the color of the smoke is mostly grey. Even though the colors may be a bit over saturated at times, we feel the method approximates the appearance of explosions rather good.

Of the two implemented turbulence methods is the Kolmogorov turbulence method the best at producing realistic turbulence. We feel the turbulence is rather convincing.

Despite pleasing results, the visual quality of the produced explosions are far from being able to fool the human eye into believing they are real. Thus, we consider requirement **R5** to be partially fulfilled.

R6: All of the above should be performed in real-time. We require a minimum of 30 frames per second, because this is usually enough to convince the human eye.

Section 7.2.1.2 shows that frame rates well above 30 can be achieved, even at rather large simulation grids. In fact, the best visual results using the 2D slice simulation are achieved at the medium sizes grids. By using the

computational power of modern GPUs, we are able to fulfill requirement **R6**.

8.3 Contributions

The method that have been proposed includes ideas and methods from previous approaches combined in new ways, some also modified to run on the GPU. Our main contributions to animating explosions in real-time can be summarized as follows:

- A physically based method for simulation of explosions, largely based on the fire simulation from [GRS06], has been proposed. It is implemented on the GPU, performing in real-time. To our knowledge, no such real-time simulation of explosions has been performed in the past.
- In contrast to [GRS06], we have included the method proposed by [FOA03] to approximate the thermal expansion that occurs during combustion. This is central to our method for simulation realistic explosions.
- The explosion simulation has been combined with the vortex particle method proposed by [SRF05] and implemented in 2D on the GPU.
- The explosion simulation has been combined with a method for internal boundaries, used by [NVi07], in a full 3D implementation. It shows that the explosion may adapt to scene geometry.
- The simulation is visualized using two separate particle systems for fire and smoke, which are rendered as textured, screen-aligned quads. The particles textures are animated, which differ from the visualization of [GRS06] where the particles textures are still images.
- A novel approach for creating animations for the textured particles is proposed. The lengths of velocity vectors from a 3D Kolmogorov turbulence field are combined with a static texture splat to produce animations that can be repeated seamlessly, as explained in section 5.3.4.
- Turbulence based on the Kolmogorov energy spectrum, as proposed by [RNGF03], is combined with the visualization phase to generate a more chaotic appearance and to reduce the symmetry caused by the volumetric extrusion. It has, to the best of our knowledge, never been used in real-time animation of explosions.

We have written a paper that describes the proposed method and the results that are achieved. The paper will be submitted to *Afrigraph 2007*.

8.4 Conclusion

We have proposed an approach for real-time animation of explosions. A fluid solver [Sta99] with vorticity confinement [FSJ01], combined with an combustion model, is used to simulate the complex behaviour of burning fuel and its gaseous exhaust products. By coupling this with the thermal expansion approximation from [FOA03], we are able to efficiently simulate deflagrative explosions. The simulation is implemented on the GPU both in full 3D with internal boundaries and in 2D. When performed in 2D, volumetric extrusion is used to implicitly define a 3D volume.

The results from the simulation are visualized using separate fire and smoke particles systems. The particles flow through the simulation domain guided by a velocity field. Additionally, their movement is controlled by an individual particle movement factor, used to create variety in how and where they move. Black-body radiation is used to calculate the color of emitted light. The particles are rendered using textured screen-aligned quads. These textures are animations that are based on Kolmogorov turbulence.

Two methods for introducing turbulence to the animation are included: The vortex particle method is used explicitly in the simulation, introducing rotational motion to the fluid, while turbulence based on a Kolmogorov energy spectrum is used to introduce turbulence to the visualization step, independently of the simulation.

We have either fulfilled or partially fulfilled the requirements that were listed in the beginning of the thesis. Although the produced explosions can not be confused with real explosions, we feel that both the appearance and motion are visually pleasing. We have shown that the physically based approach is suitable for dynamic simulation of explosion in real-time, which may prove as a useful addition to game environments.

The main reason that our explosion animation performs at such high frame rates is the parallelism and computational power of modern programmable GPUs. The continuous development of such graphics hardware will definitely enable game developers to include increasingly realistic simulation in their productions. We feel certain that physically based animation of explosions that are realistically included in games will be commonplace in few years.

8.5 Future work

The proposed method for animation of explosions can be improved in several ways. These are our suggestions towards further work:

- The method proposed by [FOA03] use fire and smoke particles as an active part of the simulation and does not model fuel and exhaust gas in scalar fields. They do, however, need a temperature field. As only four field values are needed (three velocity components and one temperature value), we suggest using a single texture on the GPU to hold these. This way the diffusion of temperature and projection of velocity can be performed simultaneously in a single pixel shader, and perhaps halving the computational load used to solve the linear equations.
- Particle systems as they are utilized in this thesis has several disadvantages. To ensure that enough particles are positioned where we would want them, a vast amount must be placed in the domain. This is a rather brute force and naive solution. A better option would be to create new particles on the fly, based on where they are needed. We suggest using geometry shaders on the GPU to dynamically create particles where combustion occurs, perhaps inspired by idea from [FOA03] and [Mic05].
- It could be looked into whether the use of the Back and Forth Error Compensation and Correction (BFECC) used by [KLLR05] could be used to reduce the numerical dissipation associated with the advection step of the Stable Fluids solver. It has, to our knowledge, not been applied to explosion simulations, thus it is interesting to investigate its effect. A publicly available GPU implementation of the BFECC method can be found in [NVi07].
- The explicit early-Z culling method proposed by [STM06] can be included in the fluid solver algorithm to either reduce the simulation time or to increase the accuracy of the projection step.
- The vortex particle method, which is only used in 2D in this thesis, could be implemented in 3D. We suggest using the shader model 4.0 functionality (i.e. Direct3D 10) to extrude points into cubes that are rendered with blending into an existing 3D velocity texture.
- As we have currently only implemented internal boundaries in the full 3D simulation, it could be investigated how they can be included into the volumetric extrusion method in a sensible way. Furthermore, simple 2D simulations (without volumetric extrusion) are useful when

the simulated explosions are far away from the camera. Thus, internal boundaries could be included in these 2D simulations to enable scene geometry to affect the simulation.

- We have not focused effort into volume rendering in this thesis, but it is still interesting to investigate if it can compete with particle based methods. We suggest looking into how ray casting, combined with physically based methods for calculating emission, absorption, and scattering of light, can be used to produce realistic results (perhaps inspired by approaches like [NFJ02] or [USKS06]).

- The internal boundaries of our current implementation are static during the simulation. In a production environment it is probably preferable to create the boundaries on the fly based on scene geometry near the explosion event. We suggest rendering scene geometry to an obstacle texture as performed by [NVi07], who also includes the speed of moving boundaries.

- In the current implementation, smoke particles are not rendered in back-to-front order. This causes visual artifacts, especially when the camera is moving closely past the explosion. We suggest sorting the particles on the GPU based their distance to the camera. Sorting algorithms suitable for GPUs has been proposed by [KSW04, Lat04]. Note that the sorting algorithm can be spread out over several frames, thus reducing the extra computational load per frame.

- To realistically include explosions in games, it generally not sufficient to let scene geometry affect the simulation as performed in this thesis. Obviously, the explosion itself needs to affect the scene as well. Methods for simulating shock wave forces could be investigated, and perhaps coupled with a physics engine like [AGE07] to apply these forces to movable objects within the scene. Approaches for simulating forces caused by explosions are presented by [YOH00, MMA99, MBA01, NF99] and can provide background for a suitable approach.

- If a game developer wants to use several explosions at the same time at large distances, using multiple instances of the current implementation would not be preferable due to the computational load. Thus, it could be investigated whether level-of-detail algorithms, such as the one proposed by [TG07], could be used to speed up the animation when explosions are placed far away from the camera.

- The simulations require numerous parameters to be adjusted. This can be difficult and is a well-known disadvantage of physically based approaches [LF02]. To ease the adjustment, we suggest removing many of the parameters by inserting real data from known substances. For

instance, if an explosion occurs because kerosene is ignited, many parameters can be automatically inserted based on actual scientific data.

- As discussed earlier, the thermal expansion can only be increased to a certain degree. We also discussed a possible explanation that involve what happens when the advection operation uses high velocities. Instead of performing one advection operation with a large time step each frame, we suggest using several advection steps with smaller time steps. This way, velocities will not be traced that far back in time. Furthermore, we expect multiple advection steps to improve accuracy of the fluid solver because piece-wise linear steps possibly approximate the non-linear Navier Stokes equations better than a single large linear step. Finally, we do not expect this to have a large impact on performance. Using two, or maybe even four, advection steps instead of one are still computationally cheap when compared to solving the systems of linear equations.

Bibliography

[AGE07] Ageia PhysX, 2007. http://www.ageia.com/, [Online; accessed 12-June-2007].

[AMH02] Thomas Akenine-Muller and Eric Haines. *Real-Time Rendering*. A. K. Peters, 2002. ISBN 1568811829.

[Cas00] Kenneth L. Cashdollar. Overview of dust explosibility characteristics. *Journal of Loss Prevention in the Process Industries*, 13:183–199, 2000.

[CM93] Alexandre J. Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer-Verlag, New York NY, 1993.

[Cra04] M. S. Cramer. Foundation of fluid mechanics, 2002–2004. http://www.navier-stokes.net/ [Online; accessed 02-April-2007].

[Dav97] Lars Davidson. An introduction to turbulence models. Technical report, Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Göteborg, Sweden, 1997.

[DL03] Todd F. Dupont and Yingjie Liu. Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. *J. Comput. Phys.*, 190(1):311–324, 2003.

[Eck06] Rolf K. Eckhoff. Differences and similarities of gas and dust explosions: A critical evaluation of the european 'atex' directives in relation to dusts. *Journal of Loss Prevention in the Process Industries*, 19:553–560, 2006.

[EV06] Lars Andreas Ek and Rune Vistnes. Towards a framework for physically based simulation of explosions in real-time. Master's thesis, Norwegian University of Technology and Science, 2006.

[FJ97] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical report, Cambridge, MA, USA, 1997. http://www.fftw.org/.

[FM97] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[FOA03] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. In *Proceedings of ACM SIGGRAPH 2003*, pages 708–715, August 2003.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[GDN97] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[GRS06] Odd Erik Gundersen, Samuel Rødal, and Geir Storli. Physically based simulation and visualization of fire in real-time using the gpu. In *Eurographics UK Chapter Proceedings: Theory and Practice of Computer Graphics 2006*, pages 13–22, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[Har04] Mark J. Harris. Fast fluid dynamics simulation on the gpu. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 637–665. Addison-Wesley Professional, 2004.

[Hol03] Thorsten Holtkamper. Real-time gaseous phenomena: a phenomenological approach to interactive smoke and steam. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 25–30, New York, NY, USA, 2003. ACM Press.

[IMDN05] T. Ishikawa, R. Miyazaki, Y. Dobashi, and T. Nishita. Visual simulation of spreading fire. In *NICOGRAPH International '05*, pages 43–48, 2005.

[Khi62] L. N. Khitrin. *The Physics of Combustion and Explosion*. Jerusalem : Israel Program for Scientific Translations, 1962.

[KKS99] D. K. Kaushik, D. E. Keyes, and B. F. Smith. Nks methods for compressible and incompressible flows on unstructured grids. In *Proceedings of the 11th Intl. Conf. on Domain Decomposition Methods*, pages 513–520, 1999.

[KLLR05] B. Kim, Y. Liu, I. Llamas, and J. Rossignac. Flowfixer: Using bfecc for fluid simulation. In *Eurographics Workshop on Natural Phenomena*, 2005.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[KW03] Jens Krueger and Ruediger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[KW05] Jens Krüger and Rüdiger Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[Lat04] Lutz Latta. Building a million particle system. In *Game Developers Conference 2004 and Graphics Hardware 2004*, 2004. http://www.2ld.de/gdc2004/.

[LF02] Arnauld Lamorlette and Nick Foster. Structural modeling of flames for a production environment. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 729–735, New York, NY, USA, 2002. ACM Press.

[LLW04] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *Pacific Conference on Computer Graphics and Applications*, pages 247–256, 2004.

[Mat97] Kresimir Matkovic. *Tone Mapping Techniques and Color Image Difference in Global Illumination*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 1997.

[MBA01] C. Martins, J. Buchanan, and John Amanatides. Visually believable explosions in real time. In *The Fourteenth Conference on Computer Animation. Proceedings, Vol., Iss., 2001*, pages 237–259, 2001.

[Mic92] Christian Michelsen. *Gas Explosion Handbook*. Gas Safety Programme, 1990-1992. http://www.gexcon.com/index.php?src=handbook/GEXHBcontents.htm, [Online; accessed 12-June-2007].

[Mic05] Microsoft. ParticlesGS, 2005. DirectX SDK (December 2005) Sample.

[Mic06] Microsoft. SoftParticles, 2006. DirectX SDK (June 2006) Sample.

[MK02] Zeki Melek and John Keyser. Interactive simulation of fire. Technical report, Texas A&M University, 2002.

[MMA99] Oleg Mazarak, Claude Martins, and John Amanatides. Animating exploding objects. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 211–218, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[NF99] Michael Neff and Eugene Fiume. A visual model for blast waves and fracture. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 193–202, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[NFJ02] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press.

[NVi07] NVidia DirectX 10 Smoke Sample, 2007. http://developer.download.nvidia.com/SDK/10/direct3d/samples.html [Online; accessed 24-May-2007].

[RE06] Erlend Randeberg and Rolf K. Eckhoff. Initiation of dust explosions by electric spark discharges triggered by the explosive dust cloud itself. *Journal of Loss Prevention in the Process Industries*, 19:154–160, 2006.

[Ree83] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[RNGF03] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 703–707, New York, NY, USA, 2003. ACM Press.

[SF93] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 369–376, New York, NY, USA, 1993. ACM Press.

[SR06] Geir Storli and Samuel Rødal. Physically based simulation and visualization of fire using the gpu. Master's thesis, Norwegian University of Technology and Science, 2006.

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Trans. Graph.*, 24(3):910–914, 2005.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[STM06] Pedro V. Sander, Natalya Tatarchuk, and Jason L. Mitchell. Early-z culling for efficient gpu-based fluid simulation. In Wolfgang Engel, editor, *ShaderX5: Advanced Rendering Techniques*, chapter TBD, page TBD. Charles River Media, Cambridge, MA, 2006.

[SU94] J. Steinhoff and D. Underhill. Modification of the Euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluids*, 6:2738–2744, August 1994.

[TG07] L. Tangvald and O. E. Gundersen. Level of detail for physically based fire. *Theory and Practice of Computer Graphics*, 2007.

[TOT+03] Daiki Takeshita, Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. Particle-based visual simulation of explosive flames. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[USKS06] Tamás Umenhoffer, Laszlo Szirmay-Kalos, and Gabor Szijártó. Spherical billboards and their application to rendering explosions. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 57–63, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[Wik] Wikipedia. Gasoline explosion. http://en.wikipedia.org/wiki/Explosion, [Online; accessed 12-June-2007].

[WLL04] Enhua Wu, Youquan Liu, and Xuehui Liu. An improved study of real-time fluid simulation on gpu. *Journal of Visualization and Computer Animation*, 15(3-4):139–146, 2004.

[WLMK02] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie Kaufman. Simulating fire with texture splats. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 227–235, Washington, DC, USA, 2002. IEEE Computer Society.

[WLMK04] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004.

[YOH00] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins. Animating explosions. In *Proceedings of ACM SIGGRAPH 2000*, pages 29–36, August 2000.

Animating Physically Based Explosions in Real-time

ABSTRACT

We present a method for real-time animation of explosions that runs completely on the GPU. The simulation allows for arbitrary internal boundaries and is governed by a combustion process, a Stable Fluid solver, which includes thermal expansion, and turbulence modeling. The simulation results are visualised by two particle systems rendered using animated textures. The results are physically based, non-repeating, and dynamic real-time explosions with high visual quality.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modelling - Physically based modelling; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Animation; I.6.8 [**Simulation and modelling**]: Types of Simulation - Animation

Keywords

Animation, Explosions, Particle systems, Real-time, Fluid Dynamics, GPU, Physically Based Animation, Natural Phenomena

1. INTRODUCTION

Thankfully, explosions are not as common in our daily lives as in computer games. In many computer games, however, they play an important role. As explosions are sudden and highly turbulent, animating visually believable and dynamic real-time explosions is a very complex task. An explosion is a rapid expansion that generates a pressure wave. This pressure wave is the main effect of an explosion. Side effects can, but do not have to, include thermal expansion, a light flash, a sudden and loud noise, a fire ball, refraction of light, smoke, flying debris, and whirls of dust. A fully realistic animation that gives life to an explosion would simulate most of these effects. However, when we use the term animation we, in fact, mean a visual simulation of a deflagration, which is a low speed explosion that propagate by transferring thermal energy to nearby combustibles. Thus, we seek to imitate the visual appearance of a high speed combustion like for instance a gasoline explosion. We do not model the pressure wave or the effects of the explosion on the environment.

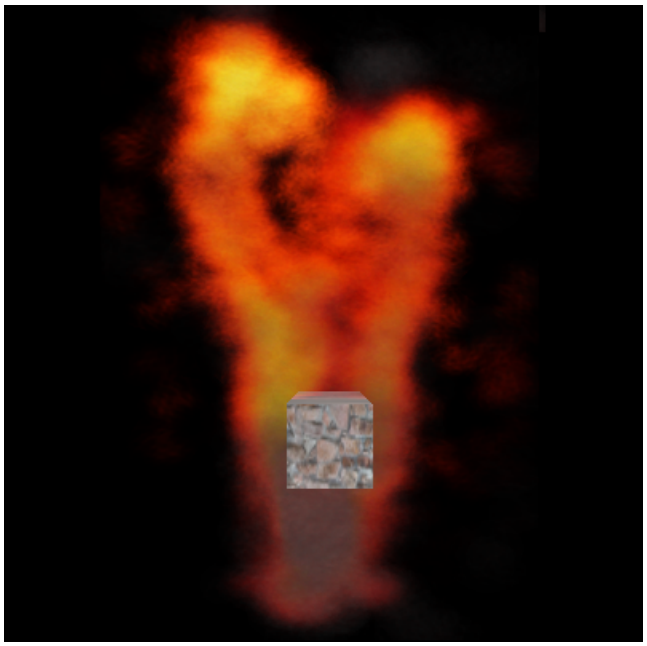

Figure 1: Full 3D explosion with obstacle interaction at 49 fps (grid size is 32x64x32). The explosion is detonated beneath a horizontal pole.

Explosions, as fire and smoke, can be simulated using computational fluid dynamics (CFD). However, as solving fluid dynamics systems are computationally demanding, there was no way to include them in real-time applications before Stam presented the well-known Stable Fluids method in 1999. This seminal paper led to several interactive and real-time techniques for animating fire and smoke. A well-known problem with the Stable Fluids approach is numerical dissipation. The effect of this numerical dissipation is not severe for effects like fire and smoke that can be kept alive by sources or external forces. For explosions that are sudden, however, numerical dissipation is more problematic as it will force the explosion to both vanish and loose its brightness

AFRIGRAPH 2007 Grahamstown, South Africa

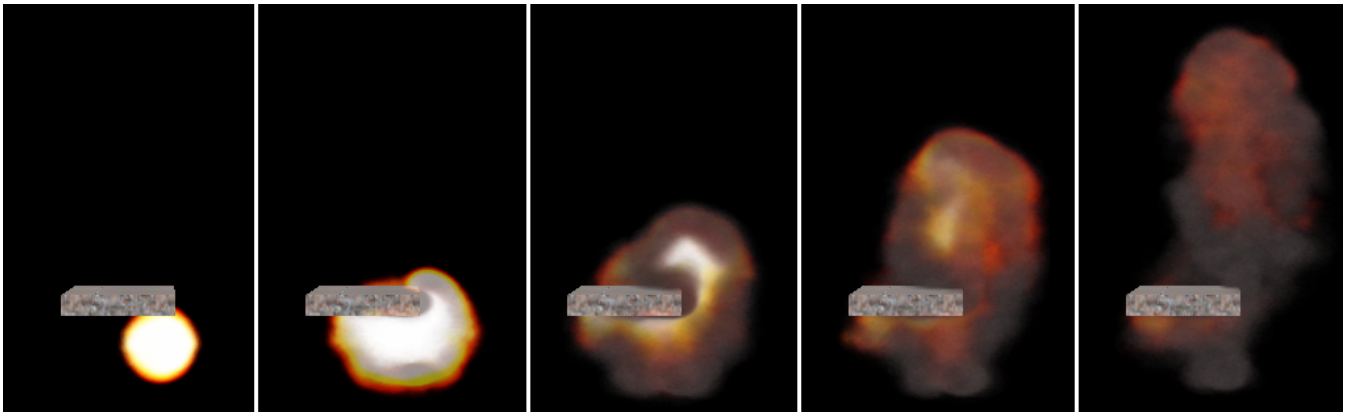

Figure 2: Five frames showing the evolution of a full 3D simulation of an explosion near a wall (31 fps).

faster than it should. This problem with using the Stable Fluids method for simulating explosions was noted by [26].

Three dimensional CFD systems used for animation evolve fluids often contained in a limited and discretized volume. The fluid contained in the volume needs to be visualised. A common technique for visualising fluid dynamics based explosions is ray-tracing the volume containing a large amount of massless particles that represent the smoke and fire parts of the explosions. This technique is still far from performing in real-time. The real-time techniques found in the literature for animating explosions are ad hoc solutions, which to a certain degree are either visually convincing or dynamic, in the sense of affecting the environment. Such ad hoc techniques include pre-calulated explosions rendered on billboards, particle systems with ad hoc heuristics [18, 25], and procedural texturing [21]. Pre-calculated explosions and solutions using procedural texturing may look nice, but they do not interact with the scene. In contrary, the ad hoc heuristics of many particle systems results in explosions that are not visually convincing, but capable of interacting with objects in the scene.

Our assumption is that animated natural phenomena look more convincing when based on the laws of nature. Fortunately, obstacle interaction follows as a consequence of basing the animation on the laws of nature. This leads us to take a physically based approach to animating explosions. In our framework, explosions are simulated using fluid dynamics together with thermal expansion and turbulence modelling. GPU implementations of stable fluid solvers are presented in [9] and [5]. Visualising the simulated explosion realistically in real-time, however, is harder. As mentioned above, realistic-looking animations of explosions are best produced using ray-tracing.

Another challenge with animating real-time explosions is small-scale detail. Small-scale detail depend on the grid size of the discretized simulation domain of the fluid simulation. Because of the real-time demands of our framework, the fluid simulation has to be done on a simulation domain with a small amount of grid cells. The small amount of grid cells, in addition to the numerical dissipation of the fluid solver used, cause small-scale detail of the explosion to disappear.

We visualise explosions using particle systems. The idea of using a particle system for visualising explosions is not a new one itself, as it was an explosion in the movie *The Wrath of Kahn* that lead to the development of particle systems [18] in the first place. In contrast to this and other ad hoc methods utilising particle systems for rendering explosions, we use a physically based fluid simulation to guide the particles. As particle systems performs well for real-time visualisation and can be fully implemented on the GPU [7], they suits our real-time requirement. To rectify the problem with small-scale detail, the particles are visualised using animated textures. Additional turbulence is added using a pre-computed Kolmogorov turbulence field to modify the velocity of the particles [17]. The rotational motion already inherent in the simulation is strengthened using vorticity confinement [1], and new rotational motion is added using vortex particles [20].

Contributions: Our main contribution is to animate explosions using a physically based method in real-time. As the offline method presented in [2], the fluid is modelled with thermal expansion, however we also simulate the combustion process as done in [12]. The simulation allows for arbitrary internal boundaries, which enables dynamic scene interaction [16], and is visualised by two particle systems. Kolmogorov turbulence is utilised to add turbulence to the trajectory of the particles [17] and to animate the texture used for rendering them [has this been done before?]. We have reached our goal of visualising physically based, non-repeating, and dynamic real-time explosions with high visual quality.

The paper is structured as follows. After the introduction, related work is presented. Then, an overview of the framework is given followed by a description of the methods used for simulating and visualising explosions. We then the present the complete algorithm and implementation details, and finally the results are discussed and we summarise and give some prospects of possible future work.

2. RELATED WORK

Methods developed in the field of CFD model the fluid motion accurately as they are used to model fire development in buildings [10] and decisions about fire safety rely on them. However, these methods scale badly as the running time depends on both the temporal and the spatial resolution of the model [4]. For methods used in computer graphics, it is enough to produce a visual convincing approximation and not a visually accurate model. Therefore, the

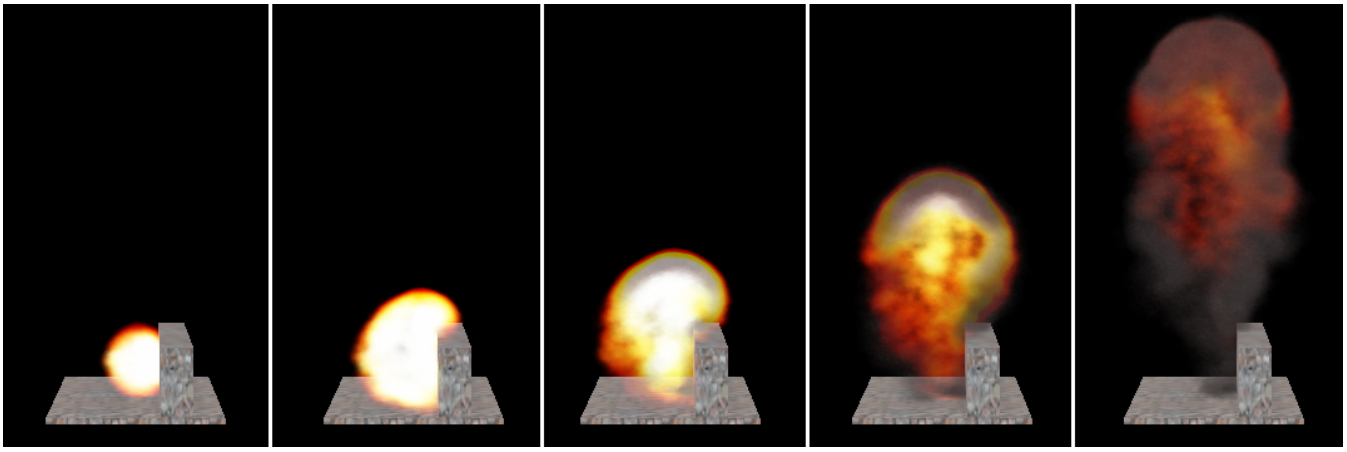

Figure 3: Five frames showing a full 3D simulation of an explosion under a flat obstacle (31 fps).

methods need not solve the fluid systems acccurately. However, when used for animation, a lot of images are needed. In motion pictures, 24 pictures are shown each second, and for real-time applications this number is even greater. Thus, it is important to minimise the production time for generating an image for both application areas.

In [4] and [3] a model for simulating fluids using the the incompressible Navier-Stokes equations on a voxel grid. The equations where solved through an explicit integration scheme whose stability required small time steps that put limitations on the usefulness of the model in real-time and interactive simulations. This limitation was removed in the unconditionally stable method presented by Stam [22]. Methods for remedying the problematic numerical dissipation have been presented and include vorticity confinement [1] and vortex particles [20]. The most promising method, however, seem to be Back and Forth Error Compensation and Correction [6].

A physically based method for animating explosions is presented in [26]. The presented method accounts for conservation of mass and energy in addition to conservation of momentum, which is modelled through the incompressible Navier-Stokes equation. The pressure is modelled explicitly and the system is capable of interacting with solids. Secondary effects include the bending of light from the blast wave. A better suited physically based simulation that use the compressible Navier-Stokes equations is presented in [2]. The focus is on modelling the flame and they do not explicitly model the pressure wave, which makes the method computationally cheaper. However, the method lacks interaction with solids. The colour of the flame is generated using Planck's law for blackbody radiation. The Stable Fluid approach is used to solve the incompressible Navier-Stokes equations for animating smoke, water and explosions in [20]. Their main contribution, however, is introducing additional vorticity using vortex particles to add detail where they exist. The method treat obstacles in three dimensions in contrary to earlier two-dimensional methods, which did not treat obstacles. A completely different method is presented in [24]. They use a discrete Lagrangian fluid model in coherence with flame and air particles. The various physical quantities used in the simulation, such as the buoyancy and the pressure gradient, are calculated based on the interaction between the particles.

While not a real-time technique when introduced, particle systems [18] are used for modelling dynamic explosions in most modern computer games. Real-time frame rates are achieved by using non-physical pressure and velocity templates to affect a Navier-Stokes based fluid simulation in [8]. Templates are used to adjust the pressure and velocity fields respectively. Using these templates it is possible to model small-scale features, and custom fluid flow can be designed as desired. Particle systems are used to render the fluids. The pressure templates need to be specified by an animator on beforehand. Thus, this method is not suitable for simulating explosions in games where you want the explosions to change dynamically according to the environment. Another non-physics method [25] use large and few spherical billboards to visualise explosions in real-time. To include the variety needed, they use real-world video clips of fire as perturbation of rendered opacity.

Other related work include [14] and [11] that model the effects of the pressure waves of explosions on the environment and physical based fire rendering methods like [12], [15], [8], and [19].

3. FRAMEWORK OVERVIEW

The explosion rendering process is divided into two parts. First, the explosion is simulated, and then the simulation is visualised. Simulation is the most computationally demanding process because it solves a fluid system. The explosion is simulated by evolving the three density fields, which are the fuel gas field, the exhaust gas field, and the temperature field, in co-evolution with the velocity field. These four fields are governed by the Navier-Stokes equations together with thermal expansion and the combustion process, which converts fuel gas to exhaust gas and heat when the temperature exceeds a certain threshold. Buoyancy due to heat then causes the hot exhaust gas to rise, while gravity pulls dense exhaust towards the ground. Vorticity confinement and vortex particles (for the 2D simulation) are used to compensate for numerical dissipation caused by the fluid solver.

After simulation, the state of the fluid system is visualised using two particle systems of animated, textured particles. Fire particles and smoke particles flow through the simulation domain guided by the velocity field and precom-

puted Kolmogorov turbulence. The particle's texture colour is computed using a black-body radiation model.

We use a voxel data structure to represent the simulation domain and will refer to each unit as a cell. Each cell contains corresponding field values. When discretizing the fields into cells, the field values are defined in the centre of the cells and assumed to be uniform inside each one, as described in [22]. There are two different kinds of cells in the simulation domain; interior cells and boundary cells.

The boundary conditions of the three density fields and the velocity field are treated differently. The boundaries of the density fields are assumed to be closed. Thus, when sampling densities outside the domain, we use the value of the closest interior cell. The boundaries of the velocity field are treated to avoid flow across them. The velocity component perpendicular to the boundary is set to the negated value of the closest interior cell. See [4] for a good treatment of boundary conditions.

4. SIMULATING EXPLOSIONS

The fluid system is solved using a stable fluid solver as described in [22]. Vortex particles [20], vorticity confinement [1], thermal expansion [2], and combustion [12] are added to the fluid simulation.

4.1 Velocity field

The velocity field $\mathbf{u}$ is governed by the Navier-Stokes equations for incompressible flow with zero viscosity, also known as the Euler equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\,\mathbf{u} - \nabla p + \mathbf{F} \tag{1}$$

The first term on the right-hand side of equation 1 is the self-advection of the velocity causing velocity to move along itself. The second term, $-\nabla p$, is the pressure gradient causing velocity to move from areas of high pressure to areas of low pressure. The last term on the right hand side of equation 1 is the external force acting on the velocity field. The external force actually consists of several separate forces as shown in equation 2:

$$\mathbf{F} = f_{vortexparticles} + f_{vorticity} + f_{gravity} + f_{buoyancy}, \tag{2}$$

where $f_{vortexparticles}$ is the result of summing the induced forces from all vortex particles, $f_{vorticity}$ is the vorticity confinement force, $f_{gravity}$ is proportional to the exhaust gas level, and $f_{buoyancy}$, which is proportional to the temperature.

When assuming the fluid is incompressible, mass conservation laws require that the velocity divergence is zero. However, during combustion the temperature of the fluid will increase and cause thermal expansion. We allow for:

$$\nabla \cdot \mathbf{u} = \phi \tag{3}$$

We let ϕ be proportional to temperature changes with a positive proportionality constant. Thus, divergence in areas where temperature is constant will still be zero. Cells where the temperature is rising, however, will have a positiv divergence. The combustion will cause the temperature to rise quickly and lead to high velocities away from the cells involved, which imitate the violent motion of explosions.

4.2 Explosion density fields

The three separate scalar fields specifying the amount of fuel gas, exhaust gas, and heat distributed throughout the simulation domain are collectively referred to as the explosion density fields. These three scalar fields are evolved by the same equation:

$$\frac{\partial d}{\partial t} = -\mathbf{u} \cdot \nabla d + \kappa_d \nabla^2 d - \alpha_d d + C_d \tag{4}$$

The parameter d is a scalar quantity that represents either the amount of fuel gas, exhaust gas, or temperature in a cell in the simulation domain; denoted by g, a or T respectively. Equation 4 describes the evolution of a scalar field over time in the simulation domain as the velocity field $\mathbf{u}$ affects the scalar field.

The first term on the right-hand side in equation 4 governs the advection of the scalar quantity d by the velocity field $\mathbf{u}$, while the second term governs the diffusion of the scalar quantity d. κ_d is the diffusion constant controlling the amount of diffusion associated with each of the density fields. Furthermore, the third term governs the dissipation of the scalar quantity d where α_d denotes the dissipation rate. The dissipation rate ensures that fuel gas, exhaust gas, and temperature will decrease over time. C_d is a combustion term that controls the effect of the combustion process on a specific density field cell. Fuel gas is combusted if the present temperature is above a certain limit. Combustion leads to a decrease of fuel gas and an increase of both exhaust gas and temperature. Predefined constants control the rate of change.

5. VISUALISING EXPLOSIONS

The simulation results are visualised by the use of two particle systems; one for fire and one for smoke. The particles move based on the velocity from the simulation step and a turbulence field calculated by the use of a Kolmogorov spectrum. Fire particles are visible where exhaust gas temperature is above a chosen temperature threshold while the smoke particles are visible where exhaust gas temperature is below the same threshold.

5.1 Particle movement

The particles are considered massless and their positions are updated based on the velocity field from the simulation step and the Kolmogorov turbulence velocity explained below. The following equation shows how to calculate the new position of the particle, $\mathbf{x}_i^{new}$, from the old position, $\mathbf{x}_i$:

$$\mathbf{x}_i^{new} = \mathbf{x}_i + \delta t(\mathbf{u}_{x_i} + \mathbf{v}_{x_i})w_i, \tag{5}$$

where δt is the time step, ($\mathbf{u}_{x_i}$ and $\mathbf{v}_{x_i}$ are the fluid velocity from the simulation step and the Kolmogorov turbulence velocity, respectively. The explosion generates high velocities, especially early in the simulation. To avoid that all the particles relocate to the outer region of the explosion during the first few moments, we introduced the denomination-less scalar w_i to create diversity in how they move. w_i can be thought of as the particle's weight or intertia, but it has no actual physical meaning.

To add more detailed movement and to avoid visual artefact caused by symmetries in the simulation or sampling, we adopt the use of a Kolmogorov spectrum to model turbulence [17]. The process explained in [23] allows the creation

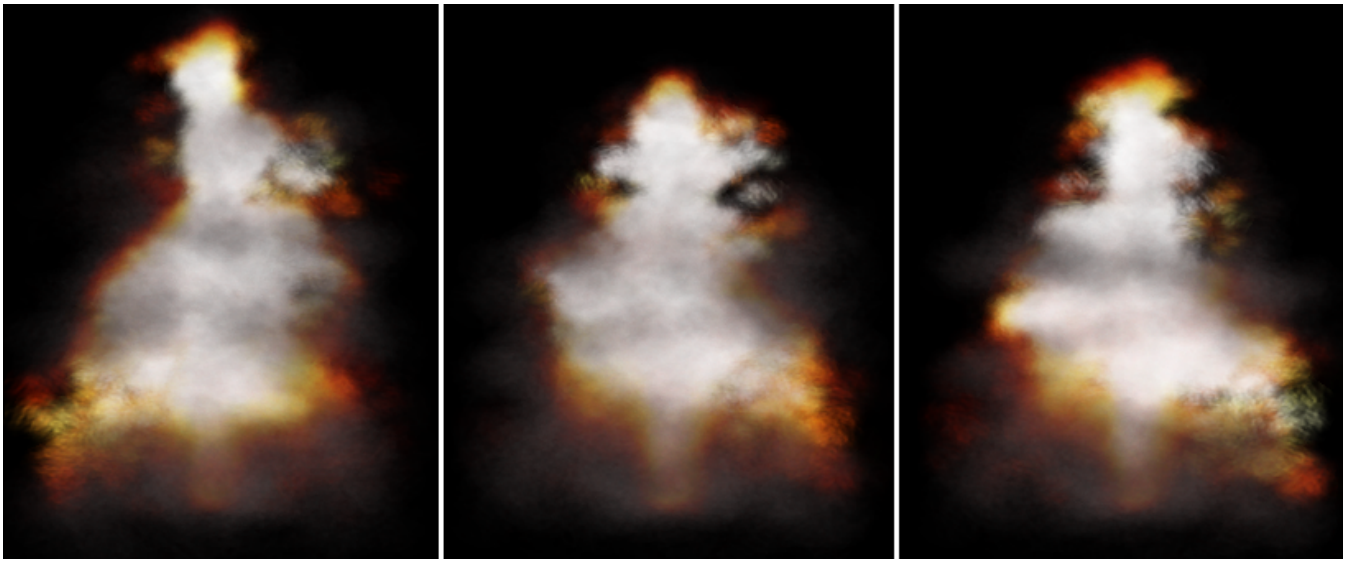

Figure 4: 2D grid slice simulations using the vortex particle method with different parameters. From left to right: Large particle count, large radii, and powerful strengths. No performance loss were observed when adding vortex particles.

of 3D velocity fields that contain small-scale turbulence. The Kolmogorov spectrum, accompanied with random numbers, is used to describe small-scale turbulence in the frequency domain. Next, an inverse Fourier transform is performed to transfer the frequency information to a useful representation in space-time domain.

The generated Kolmogorov turbulence velocity fields are periodic and may be tiled all over space. Moreover, as [17] suggested, two velocity fields are assigned to different point in time, alternating back and forth. Linear interpolation based on a time variable is used to find a velocity field at a certain point in time, and the turbulence of a given spatial position is sampled from this velocity field and added to $\mathbf{v}_i$ of equation 5.

5.2 Particle colour

As previously stated, we use two particle systems; one for fire and one for smoke. First, the fire particles are rendered, and then the smoke is blended on top. Calculation of particle colours are based on the assumption that areas of hot exhaust gas is fire and areas of cool exhaust gas is smoke. The colour for a fire particles at $\mathbf{x}_i$ is found using

$$C_{fire} = B(T_{\mathbf{x}_i}) \cdot Step(0, E_\alpha, T_{\mathbf{x}_i}) \cdot Step(0, T_\alpha, E_{\mathbf{x}_i}) \quad (6)$$

where $\mathbf{B}$ is a lookup function into a precalculated black-body radiation table, while $T_{\mathbf{x}_i}$ and $E_{\mathbf{x}_i}$ are the temperature and exhaust level at the particle's position. E_α and T_α are constants selected to define the levels of exhaust gas and temperature where fire are visualised. *Step* is a function defined as follows:

$$Step(l, u, x) = \begin{cases} 0 & \text{if x < l} \\ 1 & \text{if x > u} \\ -2(\frac{x-l}{u-l})^3 + (3\frac{x-l}{u-l})^2 & \text{else} \end{cases} \quad (7)$$

The parameters l and u define the lower and upper section for a smooth interpolation for a resulting value in [0,1].

Fire particles are blended on top of eachother using equation 8

$$C_{dst} = C_{src} + C_{dst} \cdot (1 - C_{src}) \quad (8)$$

where C_{src} is the new colour and C_{dst} is the colour allready in the render target. Using this blending operation, the result is invariant of the order, hence there is not need to render the particles in a back-to-front order.

Smoke particle colours are found by adding a small amount of colour from the black-body radiation based to a predefined grey colour:

$$C_{smoke} = Grey + \epsilon \cdot B(T_{\mathbf{x}_i}) \quad (9)$$

The constant ϵ controls how much of the colour from the black-body lookup table is used.

Next we calculate the density of the smoke and apply it to the alpha channel of its colour:

$$A_{smoke} = k \cdot (1 - Step(0, T_{max}, T_{\mathbf{x}_i})) \cdot Step(E_{min}, E_{max}, E_{\mathbf{x}_i}) \quad (10)$$

This formula ensures that smoke is not rendered where temperature is above T_{max}. Also, it ensures smoke is not rendered when exhaust level is below E_{min}, which is used to set a lower limit of when smoke is visible. E_{max} defines where exhaust levels admit the smoke to be rendered at full intensity. Smoke particles are rendered using alpha blending using

$$C_{dst} = A_{src} \cdot C_{src} + (1 - A_{src}) \cdot C_{dst} \quad (11)$$

Ideally, the smoke particles should be rendered in a back-to-front order while considering their position relative to the existing fire so that smoke behind saturated areas of fire is not rendered. However, sorting the particles is time-consuming, and thus counteract our real-time requirement.

We have found that good enough visual results are produsedwithout these considerations except when the explosion is very close to a fast moving camera.

5.3 Rendering the particles

The particles are rendered using textured, animated billboards. To create the texture animation, we use results from the Kolomogorov turbulence generator. Once again, it provides us with a 3D velocity field. We use the lengths of the vector to create a 3D scalar field scaled to the range [0,1]. Each slice in the xy-plane is interpreted as an image, i.e a 128x128x128 scalar field becomes an animation of size 128x128 with 128 images. As previously stated, the Kolmogorov generated turbulence is repeatable, so is the animation. The animation is blended with a guiding texture splat, which is different for smoke and fire. Figure 5 shows the guiding texture splat used for creating the fire particle and the first frame of the resulting animation. To increase the diversity of particle appearance each particle is also given a random orientation. The size of the particles is adjusted to the best possible visual results while achieving high frame rates.

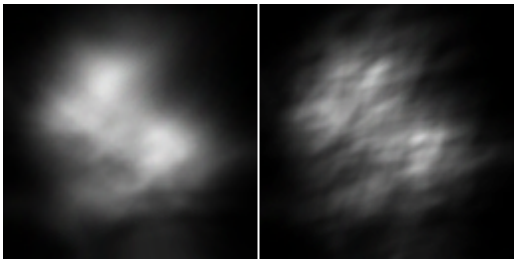

Figure 5: Fire texture splat with (left) and without (right) turbulence animation.

The colour of the fragments drawn to the screen is the multiple of the calculated particle colour and the intensity level from the texture, resulting in a visual resulting resembling small-scale turbulence. Each particle will start its animation at a random frame and change frames at a fixed animation rate.

6. THE ALGORITHM

Both the simulation and visualisation are performed each frame. The explosion simulation evolves the density fields and velocity field based on the combustion process and the stable fluid solver, which incorporate thermal expansion. Each step of the simulation is performed for all the cells in the domain. The complete algoritm is shown below:

Simulation

1. Compute velocity forces and add them to the velocity field.

2. Compute changes of density and add them to their corresponding field values.

3. Compute the thermal expansion scalar field.

4. Self-advect and project the velocity field based on equations 1 and 3 using the stable fluids solver, with thermal expansion from step 3 as ϕ.

5. Advect the denisties, applying the newly calculated velocity from step 4, and diffuse the three explosion density fields based on equation 4 using the Stable Fluids solver.

Visualisation

1. Move all fire and smoke particles based on equation 5 using the velocity field from the simulation step, the precomputed Kolmogorov turbulence velocity, and the particle's movement factor.

2. Render fire particles to the screen based on its position. Use equation 6 to find the colour and blend using equation 8.

3. Render smoke particles to the screen based on its position. Use equation 9 and 10 to find the colour and blend using equation 11.

7. IMPLEMENTATION DETAILS

To implement the fire simulation and particle simulation on the GPU, we use Direct3D 10. Textures are used to store both the main velocity and fire density fields while vertex buffers are used to hold particle data (positions, movement factor, and a random number). Computations on textures are performed using HLSL fragment shaders and some parts of the simulation uses multiple simultaneous render targets.

We implement two versions of the algorithm: One performs a two slices of a 2D simulation and combines them using volumetric extrusion [17]; the the other is a full 3D simulation with arbitrary internal boundaries inspired by [16]. The latter uses 3D textures to hold simulation data, rendered to using Direct3D 10 functionality.

Sampling from the computational domain using the particle positions requires two different approaches when the computational domain is represented as respectively a full 3D voxel volume or two 2D grid slices. When sampling from the full 3D voxel volume, simple trilinear interpolation can be used. Sampling from the 2D grid slices is more complicated though. Cylindrical interpolation [17] is used between the two slices and then bilinear interpolation is used within the two slices.

Particle movement is performed on the GPU before the updated data is streamed out to a second vertex buffer without being rendered. We alternate between these two buffers to avoid using the CPU or rendering particle data to textures. Next, particles are rendered as view aligned quads, which are generated by a geometry shader based on the particles position after a modelview projection transformation.

Initially particles are given a random position inside a sphere, which surrounds the starting point of the explosion. Each particle is also given a movement factor selected randomly from a predefined distribution. To ensure that the first few simulation steps are not saturated with smoke, we fade the smoke into the scene after a short while.

8. RESULTS AND ANALYSIS

In this section, the performance of our algorithm is evaluated and it is compared to other approaches for visualising explosions. Finally, the limitations of the algorithm are discussed.

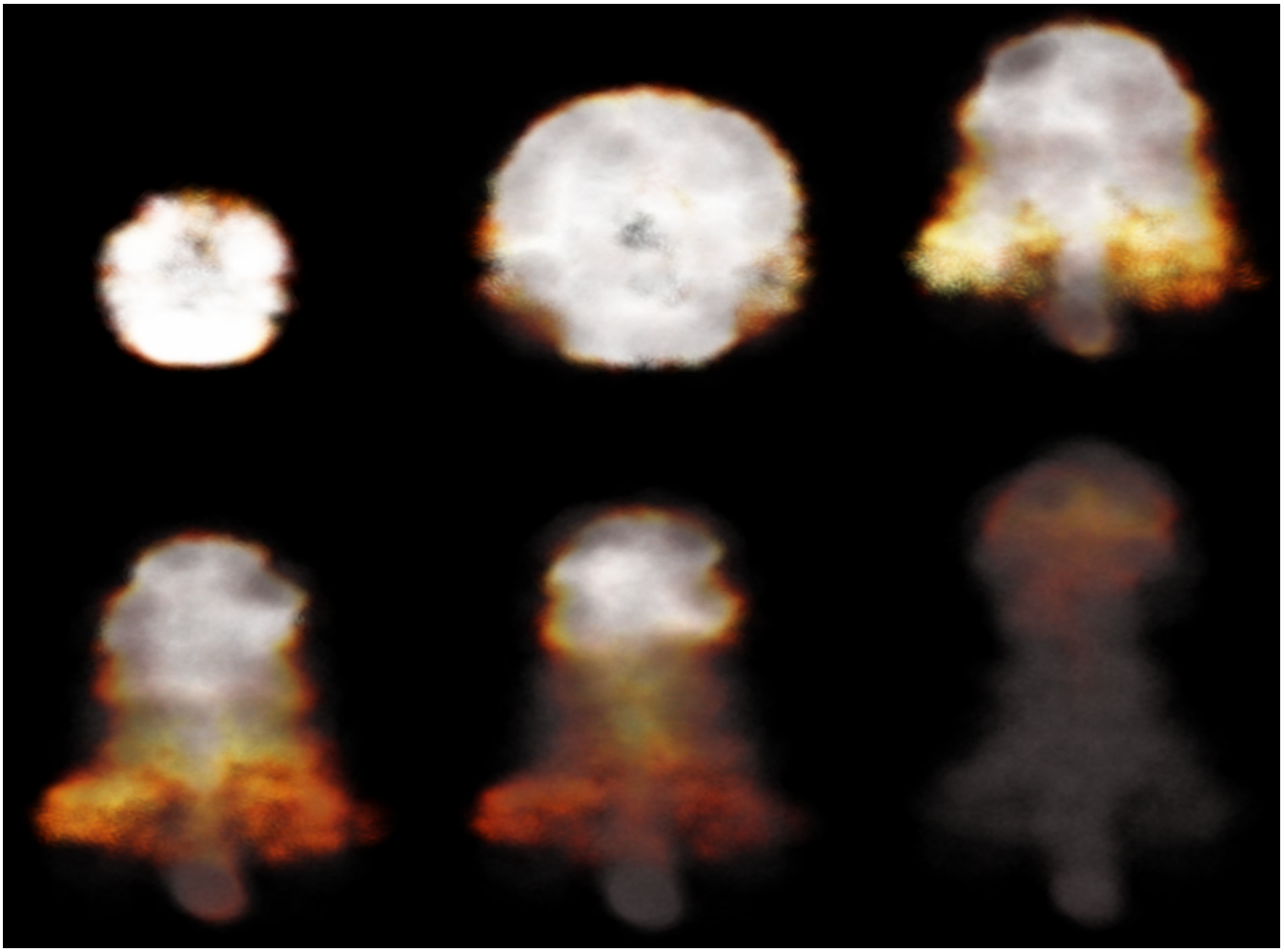

Figure 6: Six selected screen captures of the evolution of a simulation on a 128x256 grid (149 fps).

8.1 Performance and visual results

The performance of the 2D slice simulation is not directly comparable to the 3D simulation because of the latter's ability to include arbitrary internal boundary conditions, which require a substancial amount of additional shader code. All the tests were run on a 3 GHz Intel Pentium 4 CPU with 2048 MB RAM and a NVIDIA GeForce 8800 GTX with 768 MB RAM. The performance of the 2D slice simulation with and without a visualisation of 10000 fire particles and 13000 smoke particles is shown in table 1.

Grid size	Simulation only	With visualisation
64x128	768 fps	170 fps
128x256	592 fps	149 fps
256x512	184 fps	95 fps
512x1024	50 fps	31 fps
1024x2048	13 fps	11 fps

Table 1: Frame rates for 2D grid slice explosion simulation, both with and without a particle system with a total of 23000 particles.

Table 2 shows the performance of the full 3D simulation including internal obstacles, also here visualised with 10000 fire particles and 13000 smoke particles.

As expected with the 2D simulation, with the exception of the smallest grid size where the bottleneck no longer is the pixel shader, the performance of the simulation frame rates are proportional to the grid sizes. This is not the case for the 3D simulation. Performance when using 3D-textures as render targets have a bottleneck other than the pixel shaders.

Obviously, the 2D simulation outperforms the one in 3D when it comes to frame rates, but the differences of the visual results are not that clear. As shown by figure **??**, a grid size of 64x128 is sufficient for achieving visually pleasing results. However, the full 3D obstacles simulation has advantages such as being able to simulate non-symmetrical explosions and interaction with obstacles.

We have experienced with particle counts and sizes. When decreasing the number of particles, the sizes must be increased accordingly to avoid poor visual results. The number of particles itself does not affect performance noticeable, but the total number of pixel shader invocations does. In fact, having 23000 small sized particles performs better than, 13000 particles whose sizes also have been increased.

Grid size	Simulation only	With visualisation
16x32x16	209 fps	111 fps
32x64x32	52 fps	32 fps
64x128x64	13 fps	9 fps
128x256x128	1.7 fps	1.4 fps

Table 2: Frame rates for full 3D explosion simulation, both with and without a particle system with a total of 23000 particles.

When using large simulation domains, the well-known rolling motion of explosions is not as obvious as when using smaller simulation domains.

Vorticity confinement enhance small-scale rotation, and when using large simulation domains, the small-scale rotation does not enhance the large-scale rolling motion. In smaller simulation domains the vorticity that vorticity confinement enhance is not longer small, but an important part of the overall velocity. Thus, the effect of vorticity confinement on the large scale rolling motion decrease with increased simulation domain resolution.

8.2 Comparison with other approaches

There are not much other research towards real-time explosions. We compare our results to four others; two off-line methods are used as reference, a fluid guided ad hoc method [8], and the spherical billboard method of [25].

Both offline methods [26] and [2] produce very good visual results. The visual results of our method does not have the high amount of detailed small-scale variations as we use textures as an imitation, nor is the well known large-scale rolling motions as evident. In contrary to Animating Explosions our method models smoke creation as a part of the combustion process.

Compared to other real-time explosions such as the ones produced by [8], both the motion and appearance of our method look better. Additionally, [8]'s method requires an animator to create pressure templates that guide the simulation. They do not have internal boundaries or a combustion model, hence our method should be easier to include dynamically in a scene. Finally, the method used by [25] to produce explosion generate appealing pictures, but does not have a physically based simulation, hence may be difficult to include in arbitrary scenes too.

8.3 Limitations

Our method for animating explosions has several limitations. The computationally cheap 2D slice simulation is limited to symmetric explosions. Internal boundaries could be included, but again the usefullness is restricted to boundaries that are symmetrical. However, in many situations a complete 3D rendering is overkill. For instance, there is no practical difference in visual quality between a 2D or a 3D rendering when viewed at a distance.

Visualising an explosion through a particle system has its limitations. Foremost, the parts of the explosion where there are no particles will not be visible. A large amount of particles is needed to visualise the explosion. Still, there is no way to ensure that the complete simulation is visualised as the velocity field moves the particles away from the areas of the explosion with the highest velocity. Also, all particles are rendered; even those outside of where the explosion is defined by the simulation.

Finally, visual artefacts are introduced when particles are near the scene geometry. Soft particle techniques [13] may be used to solve this problem, however they come with an additional computational costs.

9. SUMMARY AND FURTHER WORK

We have presented a method for animating explosions completely on the GPU. The simulation allows for arbitrary internal boundaries and is governed by a combustion process, a Stable Fluid solver, which includes thermal expansion, and turbulence modeling. The simulation results are visualised by two particle systems comprised of textured and animated particles.

A natural extension of this work is to make the arbitrary boundaries dynamic, so that explosions could be detonated dynamically in the scene as in [16]. Now the, boundaries must be hard coded on beforehand. Another extension would be to research how to better visualise the explosion using particle systems. Now, we need many particles in order to ensure that there are enough particles where the explosion exists. However, it would be more sensible to add particles where the explosion is located at the current moment, as the explosion (soot and temperature) changes location with time. Finally, we would like to implement back and forth error compensation and correction [6] as we believe that it would work well for explosions.

10. REFERENCES

[1] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[2] B. E. Feldman, J. F. O'Brien, and O. Arikan. Animating suspended particle explosions. In *Proceedings of ACM SIGGRAPH 2003*, pages 708–715, Aug. 2003.

[3] N. Foster and D. Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, 1996.

[4] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[5] M. J. Harris. Fast fluid dynamics simulation on the gpu. In R. Fernando, editor, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 637–665. Addison-Wesley Professional, 2004.

[6] B. Kim, Y. Liu, I. Llamas, and J. Rossignac. Flowfixer: Using bfecc for fluid simulation, 2005.

[7] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[8] J. Krüger and R. Westermann. Gpu simulation and rendering of volumetric effects for computer games

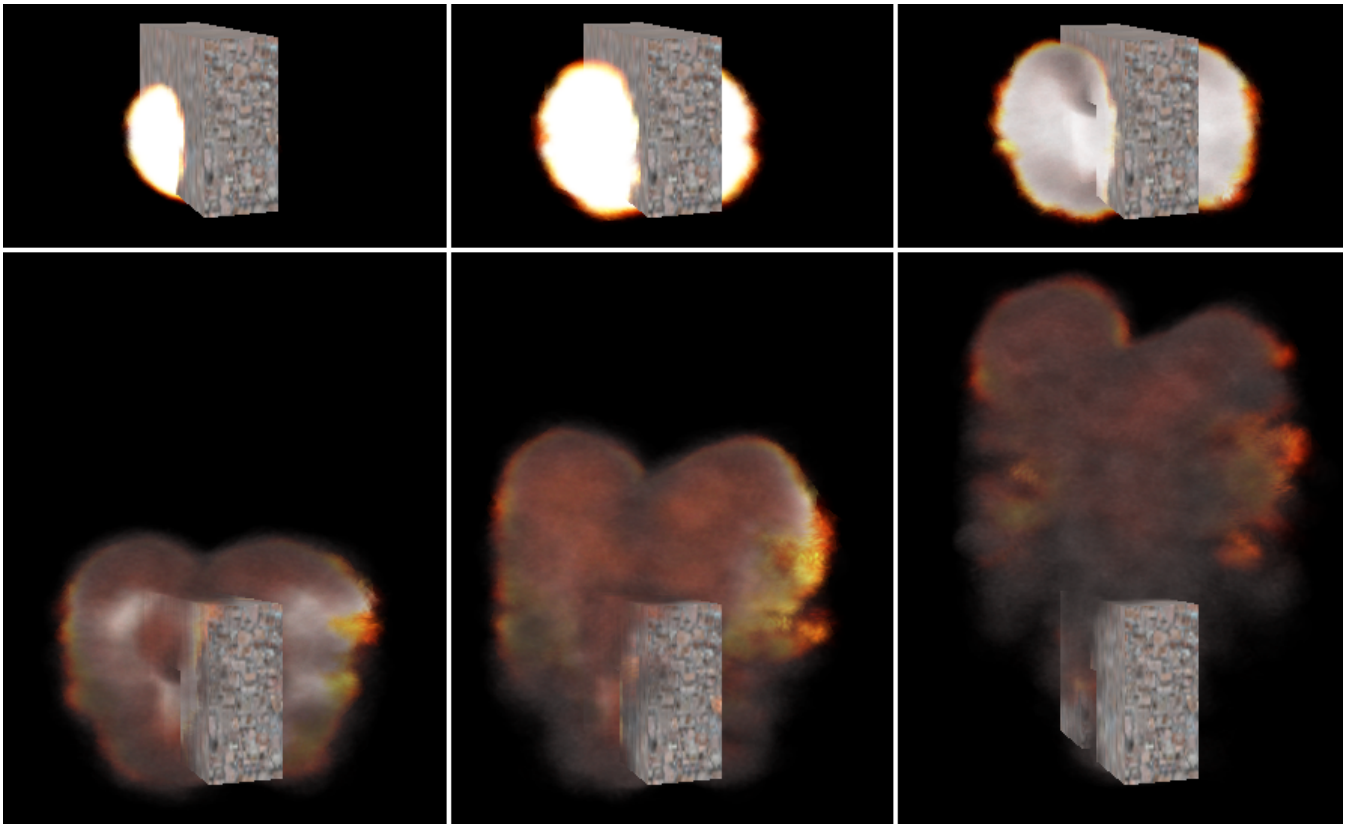

Figure 7: Six frames showing the evolution of full 3D simulation of an explosion inside an immobile arch (31 fps).

and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[9] Y. Liu, X. Liu, and E. Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *Pacific Conference on Computer Graphics and Applications*, pages 247–256, 2004.

[10] S. m. Olenick and D. J. Carpenter. An updated international survey of computer models for fire and smoke. *Journal of Fire Protection Engineering*, 13(2), 2003.

[11] O. Mazarak, C. Martins, and J. Amanatides. Animating exploding objects. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 211–218, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] Z. Melek and J. Keyser. Interactive simulation of fire. Technical report, Texas A&M University, 2002.

[13] Microsoft. SoftParticles, 2006. http://msdn2.microsoft.com/en-us/library/bb172449.aspx.

[14] M. Neff and E. Fiume. A visual model for blast waves and fracture. In *Graphics Interface*, pages 211–218, 1999.

[15] D. Q. Nguyen, R. Fedkiw, and H. Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press.

[16] Nvidia directx 10 smoke sample, 2007. http://developer.download.nvidia.com/SDK/10/direct3d/samples.html.

[17] N. Rasmussen, D. Q. Nguyen, W. Geiger, and R. Fedkiw. Smoke simulation for large scale phenomena. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 703–707, New York, NY, USA, 2003. ACM Press.

[18] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[19] S. Rødal, G. Storli, and O. E. Gundersen. Physically based simulation and visualization of fire in real-time using the gpu. In *Eurographics UK Chapter Proceedings: Theory and Practice of Computer Graphics 2006*, pages 13–22, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[20] A. Selle, N. Rasmussen, and R. Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Trans. Graph.*, 24(3):910–914, 2005.

[21] J. Spitzer. Real-time procedural effects, 2003. [Accessed online 26.10.2006].

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

[23] J. Stam and E. Fiume. Turbulent wind fields for

gaseous phenomena. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 369–376, New York, NY, USA, 1993. ACM Press.

[24] D. Takeshita, S. Ota, M. Tamura, T. Fujimoto, K. Muraoka, and N. Chiba. Particle-based visual simulation of explosive flames. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[25] T. Umenhoffer, L. Szirmay-Kalos, and G. Szijártó. Spherical billboards and their application to rendering explosions. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 57–63, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[26] G. D. Yngve, J. F. O'Brien, and J. K. Hodgins. Animating explosions. In *Proceedings of ACM SIGGRAPH 2000*, pages 29–36, Aug. 2000.