NTNU
Innovation and Creativity

# Physically Based Simulation and Visualization of Fire in Real-Time using the GPU

**Knut Erik Samuel Rødal**
**Geir Storli**

Master of Science in Computer Science

| | |
|---|---|
| Submission date: | June 2006 |
| Supervisor: | Torbjørn Hallgren, IDI |
| Co-supervisor: | Odd Erik Gundersen, IDI |

Norwegian University of Science and Technology
Department of Computer and Information Science

## Problem Description

Fire is a complex visual phenomenon, and there is a lot of room for improvement with regard to current real-time approaches for visualization of fire.
Recently, increased programmability and processing power of GPUs allows real-time simulation of fire, smoke, water, and other natural phenomena by modeling the underlying physical processes.

The task is to achieve realistic 3D fire in real-time utilizing the GPU. A physically based simulation should be developed to model the physics of fire, and the results from this simulation should be visualized using suitable approaches. The resulting fire should be suitable for inclusion in virtual environments. Focus should be both on computational efficiency and visual quality.

Assignment given: 20. January 2006
Supervisor: Torbjørn Hallgren, IDI

# Abstract

Fire is a powerful natural effect which can greatly enhance the immersion of virtual environments and games. In this thesis we describe the theory and GPU implementation of a physically based approach for simulating and visualizing 3D fire in real-time. Previous approaches are generally lacking either in visual quality, turbulence and flickering, or flexibility and extensibility. We attempt to address all these issues by using an underlying fluid simulation, modeling the mass and heat transfer aspects of the physics of fire, in combination with an explicit combustion process. The fluid simulation is used to control the behavior of a velocity field governing the motion of fuel gas, hot exhaust gas, and temperature fields, and the combustion process models the conversion of fuel gas to exhaust gas when the temperature is above the ignition temperature of the fuel gas. The velocity field is among other affected by vorticity confinement, causing a more turbulent and flickering fire, and a buoyancy force modeling upward motion. We perform the fire simulation both in 3D and in a set of 2D slices using volumetric extrusion to define an implicit 3D domain.

In order to achieve satisfying visual quality, we visualize the fire using a particle system of textured particles guided by the results from the fire simulation. The particle colors are based on black-body radiation from the hot exhaust gas, and the particles move according to the velocity field from the fluid simulation. A similar particle system is used to visualize the cooled exhaust gas or smoke. As an alternative to particle systems we have also implemented a volume rendering approach for visualizing fire, but it falls short both in performance and visual quality. Finally, we model dynamic illumination, approximating the illumination from the fire on the surrounding scene by a set of point lights, whose intensities are computed in a similar fashion as the fire particle colors. The point lights are either stationary positioned near the center of the fire, or set to follow the velocity field just like the particles of the fire and smoke particle systems.

Both the simulation and visualization of fire are implemented completely on the GPU, ensuring high frame rates without sacrificing visual quality. We have achieved a flickering and turbulent fire which compares favorably to previous approaches and works well in virtual environments, especially due to the dynamic illumination. The fire visualization also has realistic colors and intensity, and thus captures important elements of real fire. Our underlying physically based simulation enables us to efficiently simulate a variety of different kinds of small-scale fires, by altering a set of simulation parameters. One of our main contributions is implementing the explicit combustion process with fluid simulation on the GPU, as well as using it in combination with vorticity confinement and volumetric extrusion. Our contributions also include the dynamic illumination already mentioned, simulation domain advection, a novel method for modeling the behavior of fire as it is moved, and using time-dependent noise curves to model dynamic wind affecting the fire.

# 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 Samuel Rødal and Geir Storli during our $5^{th}$ and final year at the Norwegian University of Technology and Science (NTNU).

During the course of the project we have written and submitted a paper to *Theory and Practice of Computer Graphics 2006*. The paper was accepted by the conference and 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 June 15, 2006.

Samuel Rødal

Geir Storli

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Increased realism is an ever present demand in the field of computer graphics in general and the special effects and video game industries in particular. With the rapid evolution of modern graphics hardware, stunning real-time effects that were impossible a few years ago are now commonplace, but there is continually a desire for increased realism. Visualizations of natural phenomena can be used to increase the perceived realism of virtual environments, convincing the user to suspend their disbelief and become absorbed in the setting [Fer04].

Fire is a powerful natural phenomenon that can be used in virtual environments and games in order to increase immersion. However, fire is also one of hardest natural phenomena to realistically capture using traditional visualization approaches that are in little or no degree physically based. As fire is a very intense visual phenomenon, it should be realistically reproduced in order to convince the eye of a human observer.

In this thesis we focus on the task of utilizing modern graphics hardware in order to create realistic and real-time 3D fire for use in games and virtual environments. Our view is that the laws of nature play an important part and must be taken into consideration when simulating visually convincing fire. Thus, we take a physically based approach to the simulation and visualization of fire. This approach also reduces the need to come up with various ad hoc approximations which can be hard to justify. With the increased power and programmability of modern graphics processing units, more processing power is available for performing physically based simulations while still achieving real-time frame rates.

## 1.1 Motivation

Real-time visualization of realistic fire benefits areas such as virtual environments and games, where fire can be used to create a more immersive environment. For example, a visualization of fire could potentially be used to create a realistic fire place in the virtual reality recreation of “Sverresborg” [Alm05, Fal05], thus adding to the atmosphere and suspension of disbelief in the virtual environment.

There is still a lot of room for improvements with regard to current approaches for simulating and visualizing fire in real-time. Common approaches for real-time fire in virtual environments and games have typically been animated polygon models, animated textures, or particle systems with no physical basis. Animated textures depicting fire tend to look flat and are like animated polygon models rather inflexible and periodic, whereas particle systems which are not guided by a physically based simulation often fail to capture the turbulent and flickering characteristic motion of fire. Recently, a number of physically based approaches to real-time simulation and visualization have been developed, but they are still lacking in visual quality and often just use a fluid simulation without explicitly modeling combustion and other processes central to fire. An example is the approaches in [WLMK02] and [KW05] who use a fluid simulation with a particle system on top. They both fail to capture the flickering and turbulence of fire to a satisfying degree, and the colors and appearance of fire are neither very realistically reproduced.

The human eye is not easily fooled, so finding good methods for simulating and visualizing fire which can scale and utilize the ever-increasing power of modern graphics hardware is continually an area of interest. As virtual environments become increasingly realistic, the quality of fire effects must also improve to avoid degrading the immersion of the environment.

## 1.2 Objectives and requirements

Our main objective is to visualize realistic 3D fire in real-time using an underlying physically based simulation. We limit ourselves to small-scale fire effects, like bonfires and torches, as opposed to huge raging fires and explosions. The following list shows the requirements that we will eventually use to evaluate whether the objective has been fulfilled:

**R1:** A physically based fire simulation should guide the visualization.

**R2:** The fire simulation and visualization combined should be able to run in real-time. We therefore require a minimum of 30 frames per second (FPS). Although this limit is a bit arbitrary, 30 frames per second is typically enough to convince the human eye.

**R3:** The fire should be visually convincing and must therefore have a realistic appearance, close to that of real fire.

- The intensity (color and brightness) of the fire should be realistic.
- The fire should have realistic texture and low-level detail.

- The overall shape of the fire should be believable.

**R4:** The fire should move and behave realistically. This requires capturing the flickering and turbulent characteristics of fire.

**R5:** Smoke should be incorporated in the visualization.

- The smoke should have realistic appearance, including intensity, texture, and shape.
- The smoke should have believable motion.
- The coupling between fire and smoke should be realistic.

**R6:** The user should be able to interact with the fire in some fashion.

**R7:** It should be possible to use the fire in a virtual environment.

- The fire should be realistically incorporated in the surrounding scene.
- There should be computational power left for visualizing the virtual environment. The fire should not claim all the computational resources.

## 1.3 Approach

In order to achieve our goal of simulating and visualizing fire in real-time we will proceed as follows:

- We will look at a variety of recent approaches used for realistic offline and real-time fire simulation and visualization.
- We will develop a physically based 3D simulation approach for the physics of fire, attempting to combine the best methods and ideas from previous approaches.
- We will compare approaches for visualization of the simulation results in order to choose the one which gives the best trade-off between visual quality and performance.
- In order to fulfill the real-time requirement, we will utilize the programmability and computational power of modern graphics hardware for both the simulation and visualization of fire.

Our work will also build on the knowledge and experience gained from the project part of the course “TDT4715 Algorithm Construction, Science of Computing and Graphics, Specialization” at NTNU during the fall semester of 2005, where we developed a model for 2D fire simulation. Two screen captures from different timesteps of the 2D fire simulation are shown in figure 1.1.

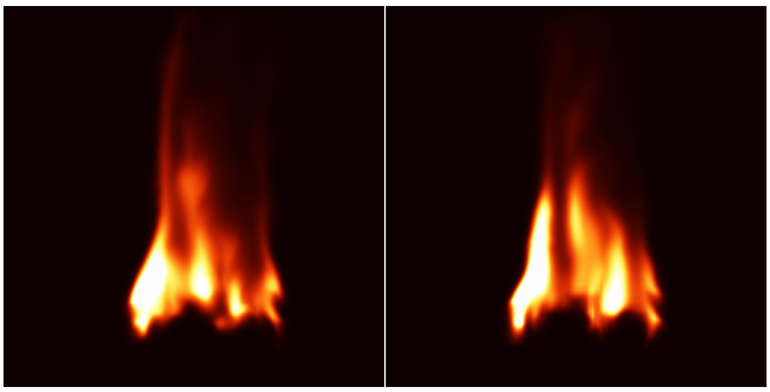

Figure 1.1: Screen captures from two timesteps of the 2D fire simulation.

## 1.4 Structure

The thesis consists of the eight main chapters described below:

**1 Introduction**
General overview, containing the motivation of the thesis, as well as presenting our objectives and requirements.

**2 Background**
Presents important background concepts, acting as a foundation for the rest of the thesis.

**3 Previous work**
Covers the state of the art in the field of simulating and visualizing fire.

**4 Simulation of fire**
Gives a theoretical overview of our real-time approach for realistic fire simulation.

**5 Visualization of fire**
Presents our main methods for visualizing the results of the fire simulation.

**6 Implementation**
Gives a detailed description of how the fire simulation and visualization are implemented, focusing on how we utilize the programmability and computational power of the graphics processing unit (GPU).

**7 Results**
Presents performance and visual results from the implementation, as well as comparing our fire with footage of real fire and with previous approaches.

**8 Discussion**
Discusses, evaluates, and concludes the thesis, as well as presenting our main contributions and giving suggestions for future work.

## 1.5 Summary

Fire is a powerful effect that can greatly enhance the immersion and suspension of disbelief of an environment. However, both physically based and traditional approaches for simulating and visualizing fire are currently lacking in visual quality and flexibility. We address these issues by attempting to use a physically based simulation to visualize realistic real-time 3D fire suitable for inclusion in virtual environments and games. We will first look at previous approaches for offline and real-time fire simulation and visualization, and then we will combine and improve previous approaches by utilizing the programmability and computational power of modern GPUs. A set of requirements will finally be used to evaluate our results.

# Chapter 2

# Background

In this chapter we aim to give an introduction to some of the areas that are important to our thesis. First, the physics of fire is described, especially the combustion process and the black-body radiation which gives fire the characteristic yellowish-orange color. Next, common applications of fire in computer graphics is discussed, followed by an overview of computational fluid dynamics (CFD). Finally, an introduction to general-purpose computation on the GPU is given.

## 2.1 Physics of fire

A suitable definition of fire is given by [Com03]: “A rapid, persistent chemical change that releases heat and light and is accompanied by flame, especially the exothermic oxidation of a combustible substance.”

There are three central aspects of fire, the chemical aspect of combustion, the physical aspect of heat transfer, and the mechanical aspect of mass transfer [BD98]:

- Combustion involves a chemical reaction, whose principal reactants are oxidant and gaseous fuel vapors.
- Heat transfer consists of diffusion, convection, and radiation. Diffusion and convection are mainly responsible for spread of fire, whereas black-body radiation gives the visual appearance we associate with fire.
- Mass transfer due to flow and buoyancy effects is responsible for the characteristic shape and motion of flames. Another mass transfer aspect is the molecular diffusion due to differences in gas composition in various parts of the flame. Molecular diffusion causes the chemical species to mix, thus allowing the chemical reactions to occur.

In the rest of this section we will focus on the combustion from the chemical aspect and the black-body radiation from the heat transfer aspect. Diffusion and convection of heat and molecular diffusion and buoyancy of mass can be modeled using computational fluid dynamics, which we present later in section 2.3. Different approaches for using computational fluid dynamics to model the physics of fire are presented later in section 3.1.

### 2.1.1 Combustion

Combustion is an oxidation-reduction reaction, where the oxidizing agent is oxidant and the reducing agent is fuel [BD98]. Fuel not already in a gaseous state needs to vaporize before it can react with the oxidant (typically oxygen) and combust. Vaporization occurs when the fuel is sufficiently heated, transforming liquid or solid fuel into gaseous fuel vapors. In addition, the fuel gas needs to reach its ignition temperature before the combustion takes place. The general equation for combustion can be written as shown in the following equation:

$$a \text{ oxidant} + b \text{ fuel} \rightarrow c \text{ CO}_2 + d \text{ H}_2\text{O} + ... \tag{2.1.1}$$

The left hand side of the equation shows the oxidant and fuel which are prerequisites for combustion. During combustion $CO_2$, water vapors ($H_2O$), and hot gaseous byproducts like carbon soot are formed. Combustion also produces more heat, which again triggers more combustion, keeping the process alive until there is not enough fuel, heat, or oxygen.

There are two main kinds of flames, diffusion flames and premixed flames. Diffusion flames occur when a pure fuel gas coming from a fuel source mixes with oxidizing gas through a mixture zone. A candle flame is a typical example of a diffusion flame. The physical phenomena involved in the candle flame are shown in figure 2.1. Once ignited, heat melts the stearin which soaks into the wick of the candle. Gaseous vapors then continuously vaporize off the wick, fueling the fire. Surrounding the vaporized fuel is the blue reaction zone where the combustion reaction occurs. Campfires and torches behave in a similar fashion as candle flames.

The second kind, premixed flames, occur when fuel and oxidizing gases are well-mixed before entering the reaction zone of the flame. The blueish flame seen in natural gas burners is a typical example of a premixed flame.

### 2.1.2 Black-body radiation

During the combustion reaction of a diffusion flame different hot gaseous products are formed. A black-body radiation emitted by the gaseous products, in particular carbon soot, gives the characteristic yellowish-orange color that we associate with fire [NFJ02].

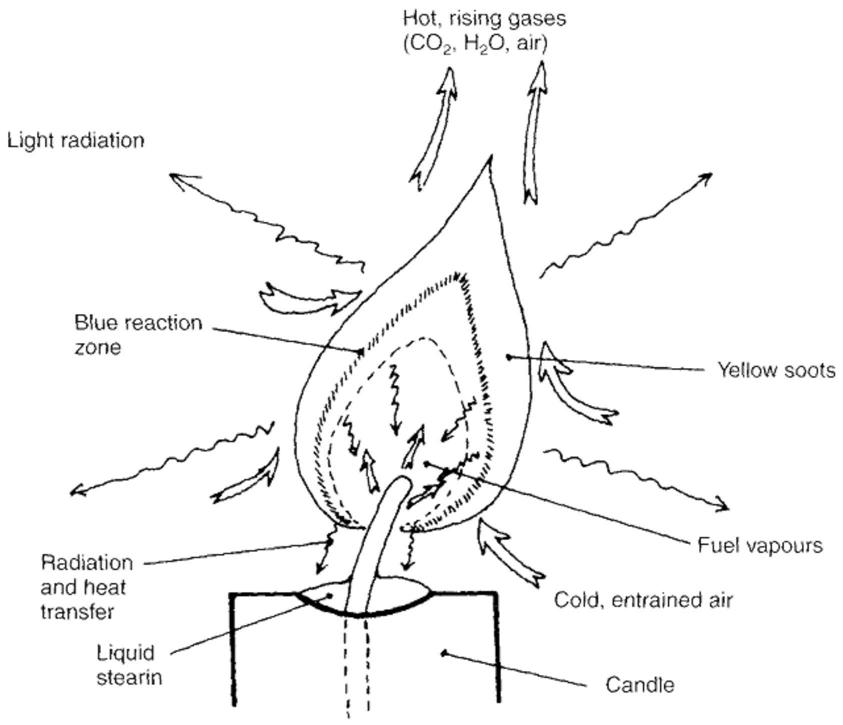

Figure 2.1: The physical phenomena involved in a candle's flame [BD98].

A lot of heat is also generated during the reaction, triggering more combustion and the creation of more hot gaseous products. The temperature will decrease some distance away from the reaction zone, resulting in a decreasing black-body radiation until the yellowish-orange color no longer is visible. When the gaseous products are sufficiently cooled, they will appear as smoke and soot.

The black-bodies emitting black-body radiation are the soot particles created before and during combustion. Planck's formula, shown in equation 2.1.2, gives the intensity of a certain wavelength $\lambda$ radiated by a black-body with temperature $T$, where $h$ is Planck's constant, $c$ is the speed of light, and $k$ is Boltzmann's constant [Pla06].

$$B_\lambda(T) = \frac{2\pi hc^2}{\lambda^5\left(e^{\frac{hc}{\lambda kT}} - 1\right)} \tag{2.1.2}$$

Figure 2.2 shows the black-body radiation spectrum as a function of wavelength. As can be seen, the emitted energy increases rapidly with temperature. The visible spectrum ranges from 400 to 700 nm, and the black-body radiation spectrum has a peak at the right end of this spectrum, which corresponds to yellow, orange, and red colors. This peak explains the yellowish-orange color that is typical for fire.

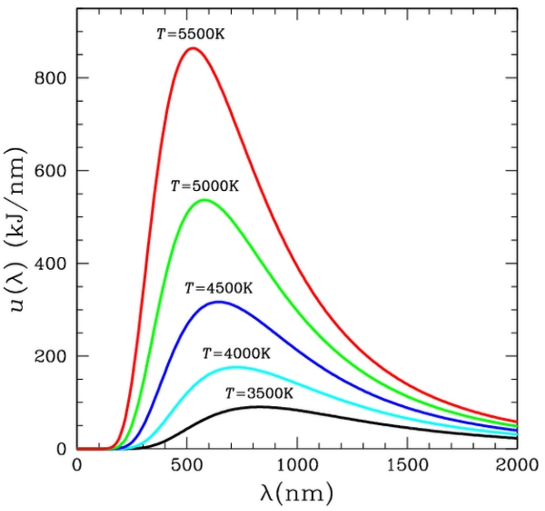

Figure 2.2: The black-body radiation spectrum as a function of wavelength.

## 2.2 Applications of fire in computer graphics

With the demand for increased realism in computer graphics for areas such as movies, computer games, and virtual environments, certain effort has been made in approaches for portraying the characteristic visual elements of fire. This section describes some of the different approaches that have been used for fire effects in movies and games.

### 2.2.1 Special effects in movies

The special effects industry has been known to push the boundaries of computer graphics (CG) to their limits. The recent animated movie *Shrek* (2001) created dragon's breath and torches as well as other fire effects by using procedural mechanisms combined with various methods of allowing animators to control high-level details like the spread of fire. The approach used in *Shrek* is further described in [LF02]. Until recently, however, relying on physical simulation has been too computationally expensive even for movies. Thus, tricks and approximations have been used to achieve fire-like behavior and appearance. An example of these approximations is the movie *Star Trek II: The Wrath of Khan* (1982), which used a huge particle system to simulate the propagation of an explosion on a planet's surface, as described in [Ree83]. As computing power increases exponentially, we predict that the use of physical simulations of fire for special effects in movies will continue to grow. Other recent movies with memorable CG based fire effects include *The Lord of The Rings: The Fellowship of The Ring* (2001) and *Reign of Fire* (2002). *The Lord of The Rings* used a particle system where each particle was an animated sprite with footage of real fire and smoke. This particle system was used to create the impressive balrog, a monster cloaked in fire and smoke, as shown in figure 2.3. NVIDIA used a similar approach for their Vulcan demo [Ngu04]. *Reign of Fire* in turn used computational fluid dynamics and volume rendering to help create fire-breathing dragons [DHR+02].

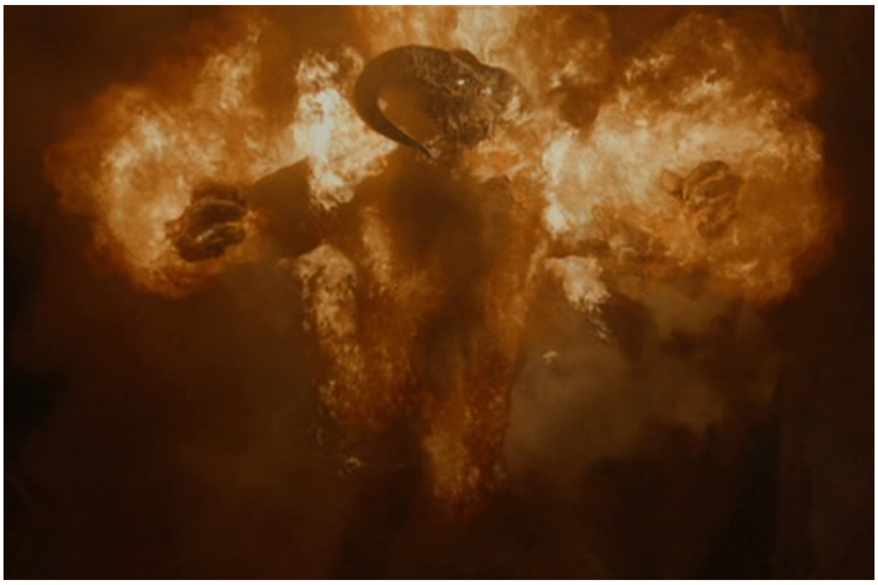
Figure 2.3: The balrog from *The Lord of The Rings: The Fellowship of The Ring* (2001).

### 2.2.2 Fire effects in games

Computer games is also a field which has experienced rapid growth in visual quality and physical correctness thanks to increased computing power. Particle systems are common in 3D game engines, and are often used to simulate and visualize effects such as flames and fireballs. Animated textures are also a simple way to get realistic, though repetitive and non-flexible fire renderings. Figure 2.4 shows a screen capture from the recent computer game *Quake 4*, showcasing one of the game's fire effects. Games have traditionally been using non-physically based methods for these kinds of effects, focusing on creating acceptable visuals that are not too computationally expensive. This might change with the advent of more powerful hardware. For example, a new physics coprocessor dubbed *PhysX* has recently been released, promising acceleration of tasks such as collision detection, fluid dynamics, and more [AGE06].

## 2.3 Computational fluid dynamics

Recently, CFD has been used increasingly in combination with computer graphics to simulate and visualize natural effects such as water, smoke, and fire. With increasing

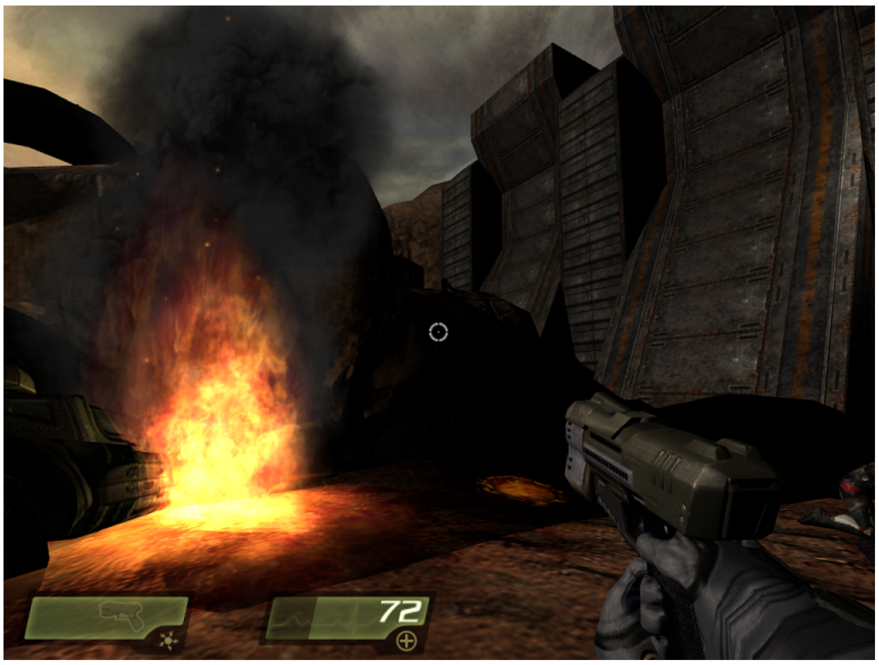

Figure 2.4: Fire effect in *Quake 4*.

processing power it has become more viable to use physically based simulations instead of ad hoc approximations for both offline and real-time graphics. As many of the current methods of simulating fire use CFD for modeling the heat and mass transfer aspects of fire, we first give a brief overview of the field of computational fluid dynamics, and then present an approach for performing stable fluid simulations in real-time, suitable for computer graphics.

### 2.3.1 Field overview

A fluid is a substance which flows under pressure, and fluid mechanics is the branch of fluid dynamics concerned with the study of fluids in motion [LL98, Bat00]. Fluid flow can be characterized by a set of aspects governing the behavior of the flow:

- Compressible flow, as opposed to incompressible flow, occurs when the pressure variation in the fluid is large enough to incur substantial changes in the fluid's density.
- Viscous flow, as opposed to inviscid flow, is used to model fluids in which fluid friction has a significant effect. The amount of fluid friction is described by the fluid's viscosity.
- Steady flow, as opposed to unsteady flow, is used when the changes of fluid properties with time is zero.

- Turbulent flow is seemingly chaotic and is caused by unsteady vortices appearing on many scales and interacting with each other. Flow which lacks turbulence is called laminar, and is seemingly smooth and stable.

CFD is the application of computers to analyze or solve problems in fluid dynamics. The field originated in the 1960's [Com06] and has since experienced large growth, thanks to the exponentially increasing power of computer hardware. Current applications of CFD range from physics research and industrial simulations to computer aided design and special effects. To be able to simulate fluid flow, the spatial domain of interest is usually discretized into a mesh or grid of small cells. Each cell contains the state of the fluid in that cell, which usually includes velocity, density, temperature, and pressure. These cell states are then updated in discrete timesteps to study how the fluid evolves over time.

Updating the cell states is typically done by solving the Navier-Stokes equations, which are capable of modeling compressible, viscous, unsteady, and turbulent flow. The resulting data can then be visualized or analyzed depending on the requirements of the application. Many different CFD algorithms exist, but common to most of them is the discretization of the computational domain into a mesh or grid, and the simulation governing the evolution of a velocity field in the domain.

### 2.3.2 Stable fluids

A stable method for solving the incompressible Navier-Stokes equations is presented by Jos Stam in [Sta99] and [Sta00]. Traditionally, work on computational fluid dynamics has been more focused on precision than efficiency. The applications have typically been in engineering and industrial fields, where the governing factor has been numerical correctness and not visual quality. Classical CFD models were unstable, meaning that the simulation accuracy deteriorates rapidly or diverges when the timestep becomes too big. This makes these models less than suitable for real-time simulation, where there is a limit to how low the timesteps can be.

The approach presented by Stam uses several steps to solve the incompressible Navier-Stokes equations with the inclusion of an external force **f**, as shown in equations 2.3.1 and 2.3.2. The variable **u** is the velocity field of the fluid, and the first equation gives the partial derivative of **u** with respect to the time $t$. The terms on the right hand side of the equation are the advection, pressure, diffusion, and force terms respectively. The second equation states that the velocity field should have zero divergence, which implies that mass should be conserved. The equations are usually defined for a bounded 2D or 3D computational domain $D$ in which the fluid lies.

$$\frac{\partial \mathbf{u}}{\partial t} = -\left(\mathbf{u} \cdot \nabla\right)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2.3.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.3.2}$$

The equations are approximated by Stam's approach in order to stably evolve the velocity field $\mathbf{u}$ over a single timestep $\Delta t$. First, the force term is added to the velocity field, as shown in the following equation:

$$\hat{\mathbf{u}}(\mathbf{x}) = \mathbf{u}(\mathbf{x}) + \mathbf{f}(\mathbf{x})\,\Delta t, \tag{2.3.3}$$

where $\hat{\mathbf{u}}$ is the velocity field after the force term $\mathbf{f}$ has been added. External forces are applied to the fluid by manually manipulating $\mathbf{f}$.

Next, the velocity field is self-advected, which can be interpreted as the velocity pushing itself forward. The following equation shows the implicit approach used by Stam to perform the self-advection:

$$\hat{\mathbf{u}}(\mathbf{x}) = \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x})\,\Delta t)\,. \tag{2.3.4}$$

An intuitive way of looking at this equation is regarding the velocity field as a set of particles, centered in each grid cell. Each particle is traced backward one timestep using its current velocity, to find the previous position the particle must have had to end up at the current position. The velocity at the previous particle position is then interpolated between the neighboring cells to get the new velocity at the current particle position. Figure 2.5 shows how a particle is traced back over a timestep $\Delta t$ and which grid cells are used for interpolating the velocity at the previous position.

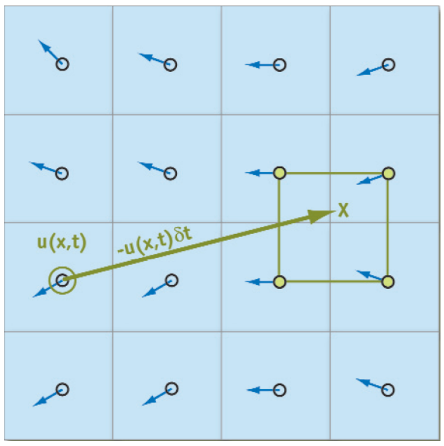

Figure 2.5: Implicit fluid advection with a timestep of $\delta t$ [Har04].

After the advection term has been evaluated, the diffusion term $\nu\nabla^2\mathbf{u}$ is added. $\nu$ is the viscosity of the fluid, and describes the fluid's resistance to movement caused by friction inside the fluid. As for advection an implicit approach is used, as shown in the following equation:

$$\left(\mathbf{I} - \nu\Delta t\nabla^2\right)\hat{\mathbf{u}}(\mathbf{x}) = \mathbf{u}(\mathbf{x})\,, \tag{2.3.5}$$

where $\mathbf{I}$ is the identity matrix. The equation is used to find a new velocity field $\hat{\mathbf{u}}$, that when diffused backwards in time yields the current velocity field $\mathbf{u}$. As can be seen, the equation yields a system of linear equations, which can be solved iteratively with among other the Gauss-Seidel or Jacobi iteration methods.

After the force, advection, and diffusion terms have been added, the mass conservation requirement in equation 2.3.2 will generally no longer be valid. However, any vector field can be uniquely decomposed into the sum of a non-divergent vector field and the gradient of a scalar field. The scalar field corresponding to the decomposition of the vector field $\mathbf{u}$ is the pressure term $p$, and by calculating it and subtracting its gradient from $\mathbf{u}$ we get the resulting non-divergent velocity field $\hat{\mathbf{u}}$, as shown by the following equation:

$$\mathbf{u} = \hat{\mathbf{u}} + \nabla p. \tag{2.3.6}$$

Calculating the pressure term $p$ is done by solving the Poisson equation resulting from multiplying each side of equation 2.3.6 by "$\nabla$", as shown in the following equation:

$$\nabla \cdot \mathbf{u} = \nabla^2 p. \tag{2.3.7}$$

Just like for diffusion, equation 2.3.7 yields a system of linear equations. When solving the Poisson equation using an iterative method, the Neumann boundary condition $\frac{\partial p}{\partial n} = 0$ should hold on the boundary of the domain, ensuring that the pressure gradient perpendicular to the boundary normal is 0.

With small modifications, Stam adapted the method for solving the Navier-Stokes equations in order to simulate the spread of scalar densities carried by the fluid's velocity field. This modification is important, as we usually do not want to visualize the velocity field directly, but rather show the effects of the velocity field on a substance like dye or smoke. The following equation gives the evolution of a scalar density field $d$:

$$\frac{\partial d}{\partial t} = -\mathbf{u} \cdot \nabla d + \kappa_d \nabla^2 - \alpha_d d + S_d, \tag{2.3.8}$$

where $\mathbf{u}$ is the velocity field, $\kappa_d$ is the diffusion constant, $\alpha_d$ is the dissipation rate, and $S_d$ is a source term. The advection term, diffusion, and source terms are solved just like for the velocity field, with $S_d$ corresponding to the force term $\mathbf{f}$ in the Navier-Stokes equations presented earlier. The difference is that instead of pushing the velocity itself forward, the advection step now pushes the scalar density field forward based on $\mathbf{u}$. Dissipation can be solved using the following equation over a single timestep:

$$(1 + \Delta t \alpha_d)\, \hat{d}\,(\mathbf{x}) = d\,(\mathbf{x})\,. \tag{2.3.9}$$

While allowing arbitrary timesteps, the stable fluids approach trades accuracy for performance. Stam's focus was not on highly accurate simulations, but on achieving acceptable results efficiently without sacrificing stability. This means that the algorithm is especially suited for visualization and special effects, where visual quality and perceived realism is the main requirement. The stable fluids approach captures the visual elements of fluid flow, while being fast enough to achieve interactive frame-rates on reasonably sized grids.

## 2.4 General-purpose computation on the GPU

The GPU on modern graphics cards has evolved into an extremely powerful and flexible processor. Fully programmable vertex and fragment processors with support for vector operations up to full IEEE floating point precision are provided by the latest graphics architectures. The GPUs are architecturally highly parallel streaming processors that are optimized for vector operations, and they deliver hundreds of gigaflops of single-precision floating-point computations, as compared to approximately 12 gigaflops for current high-end CPUs [KF05]. To make it easier to access this computational power, high-level languages for graphics hardware have been developed. Modern GPUs are fully capable of general-purpose computation, and utilizing these capabilities can result in significant performance gains over the CPU.

### 2.4.1 The graphics hardware pipeline

The graphics pipeline consists of a sequence of stages that operate in parallel and in fixed order. The input for each stage is received from the previous stage and the output is sent to the next stage. In figure 2.6, a modern and programmable graphics pipeline is expressed as a stream programming model.

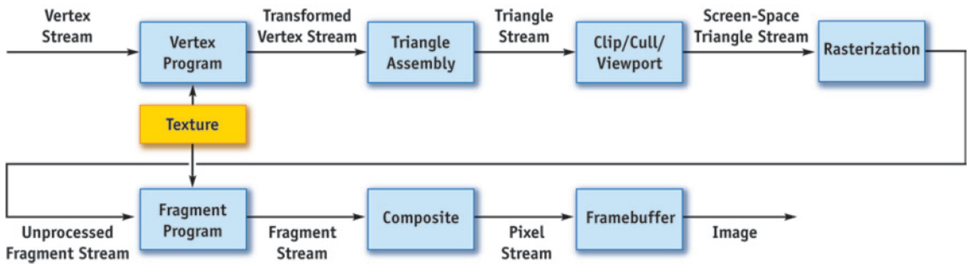

Figure 2.6: The graphics pipeline as a stream programming model [Owe05].

A stream is an ordered set of data of the same type and is operated on using kernels [Owe05]. A kernel operates on entire streams of elements, performing the same operation for all elements in the stream. The kernel takes one or more streams as input and produces one or more streams as output. In the graphics pipeline the *Fragment Program* kernel is limited to producing one output stream for each of the RGBA-channels. The outputs from a kernel operation are functions only of the input to that kernel. In addition, the computations on one stream element do not depend on the computations of other elements in the same stream. These restrictions allow stream elements to be processed in parallel using data-parallel hardware and this parallelism is heavily exploited on modern GPUs.

In figure 2.6 arrows indicate data expressed as streams and blue boxes indicate computations expressed as kernels. A stream of vertices is the input to the *Vertex Program* kernel. Each vertex usually has several associated attributes such as position, color, normal, and texture coordinates. The programmable vertex processors apply a vertex program to transform the stream of vertices. During this transformation, texture lookups can be performed using Vertex Shader 3.0 functionality. Then, the *Triangle Assembly* kernel is responsible for generating the rendering primitives, like lines and polygons, that are tessellated into triangles. After perspective transformation, clipping, and culling, the elements in the resulting data stream are localized in the screen space domain.

The *Rasterization* kernel determines the set of pixels that cover the triangles in the triangle stream and outputs a stream of fragments. A fragment can be considered a prototype pixel and contains all information needed to generate a shaded pixel in the final image. The information includes destination in the framebuffer, depth, and interpolated values of the attributes associated with the vertices from the vertex stream, such as color and texture coordinates. The fragment stream is then processed by the programmable fragment processors which apply a fragment program to each fragment. Using interpolated texture coordinates, the fragment program can perform texture lookups to be used in the fragment computation. The fragments are then combined in the *Composite* kernel before finally written to the framebuffer. To perform general-purpose computations on the GPU the programmable fragment processors are utilized with kernels written in terms of fragment programs.

### 2.4.2 Computational analogies

Because of the increased programmability of fragment processors the GPU is now capable of general-purpose computation. An overview of general-purpose computation on the GPU is given in [Har05]. Simple analogies between traditional CPU computational concepts and their GPU counterparts are given in table 2.1, and these concepts are further described in this section.

The fundamental data structure on the GPU is the texture, so when an array of data is used on the CPU, a texture containing data must be used on the GPU. By packing data into the four color channels of a texture increased parallelism is achieved, allowing computation on four streams at once. On the CPU, a loop is used to iterate over the elements of an array, each iteration performing a kernel operation on the given array

| **CPU Concept** | **GPU Concept** |
|---|---|
| Arrays | Textures |
| Inner loops | Fragment programs |
| Feedback | Render-to-texture |
| Computation invocation | Geometry rasterization |
| Computational domain | Texture coordinates |
| Computational range | Vertex coordinates |

Table 2.1: Analogies between CPU and GPU computational concepts.

element. When transforming a loop operation from the CPU to the GPU, the loop kernel is represented by a fragment program on the GPU and applied to all elements of the fragment (data) stream. The fragment program results are written to an output texture using render-to-texture functionality, and the output texture can later be used as input to another operation.

In order to invoke the computation on the GPU, a method to generate a stream of fragments is needed. This stream can be generated by drawing proxy geometry like rectangles and lines. The rasterizer then generates a fragment for each output buffer pixel covered by the geometry. Associated with the geometry-defining vertices is a set of texture coordinates, which are linearly interpolated by the rasterizer, thus generating a corresponding set of texture coordinates for each fragment. When a fragment is processed by the fragment processor, these coordinates are passed as input and can be used as array indices for performing texture fetches from the input textures. The geometry-defining vertices can be used to determine which pixels are generated, thus controlling the output range of the computation.

### 2.4.3 Invoking the computation

In figure 2.7 we show how to invoke a general-purpose computation on the GPU. The computation is performed on the data from the input texture and the results are written to an output texture. To perform the computation in the *Fragment Program* kernel, a stream of fragments must be generated. We assume the computation is to be performed on a 2D texture of size 8x8, and thus a stream of 8x8 fragments has to be generated. First, the viewport width and height are set to the dimensions of the input and output textures. Next, we specify the coordinates of four vertices of a viewport-filling quad. These vertex coordinates control the output range of the computation. Together with each vertex a texture coordinate is specified. We assume unnormalized texture coordinates in the range $[0, 0]$ to $[width, height]$, where *width* and *height* are the width and height of the input and output textures and in this case also the width and height of the viewport.

In figure 2.7 the *Geometry Processing* kernel consists of the *Vertex Program*, *Triangle*

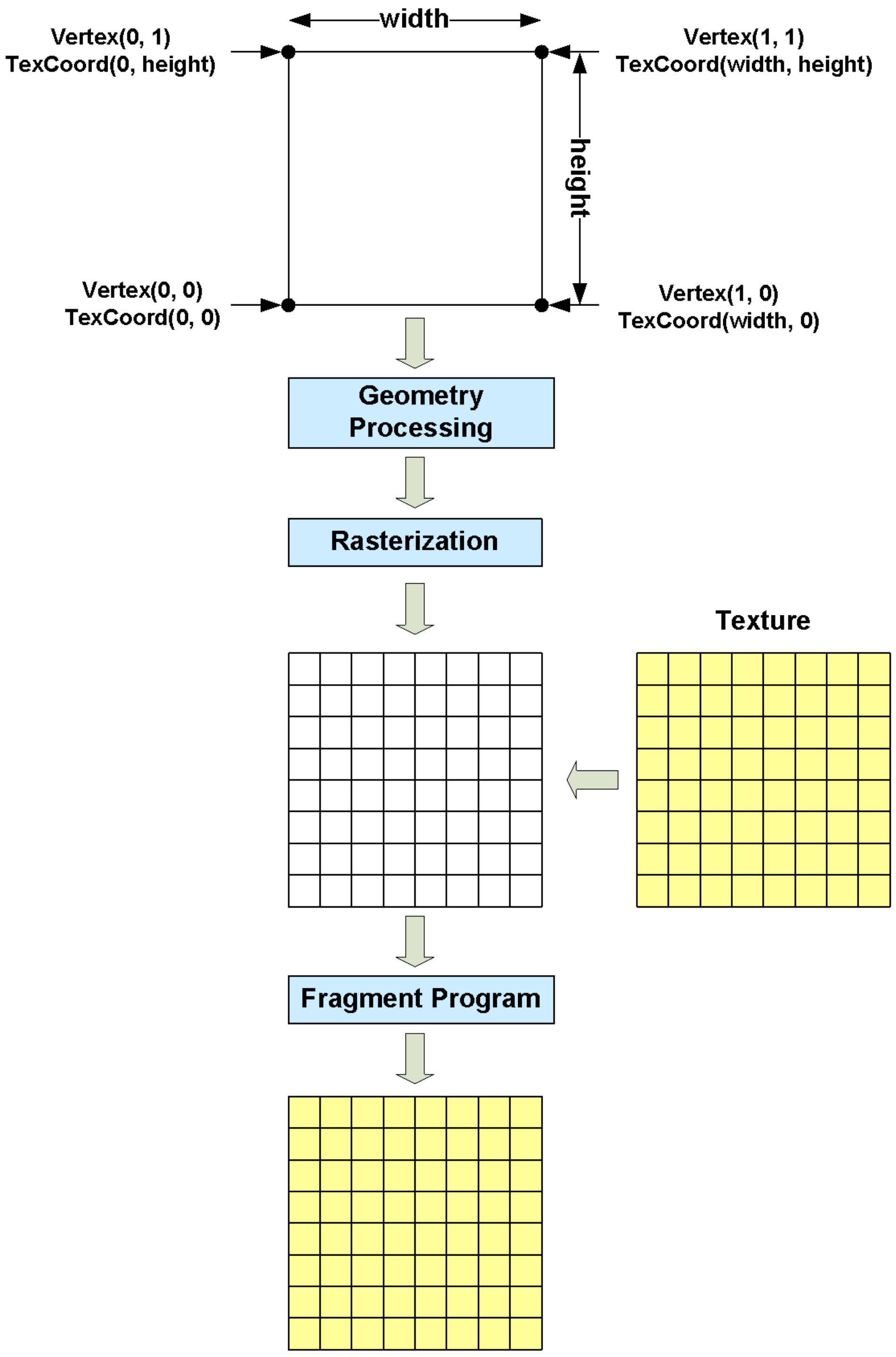

Figure 2.7: General-purpose GPU computation on a texture.

*Assembly*, and *Clip/Cull/Viewport* kernels from figure 2.6. The vertex processors must be set to simply pass the four quad vertices through untransformed, ensuring that 8x8 fragments are generated when the quad passes through the *Rasterizer* kernel. The texture coordinates from each vertex are interpolated to generate a set of coordinates for each fragment. In the *Fragment Program* kernel these coordinates are used for each fragment processed to fetch the corresponding value from the input texture. The resulting fragment value is finally written to the active render target, which can be another texture to be used later in another computation.

## 2.5 Summary

Fire is the result of a chemical reaction, where fuel gases and oxygen react due to heat, creating hot gaseous products. The hot gaseous products act as black-bodies, emitting black-body radiation, which is a function of wavelength and temperature.

Computer generated fire has been used in a large amount of games and movies. The demand for increased realism in movies has caused a shift from tricks and approximations to more physically based methods for visualizing fire. Games still use ad hoc methods like particle systems with no physical basis and animated textures to simulate fire effects like flames and fireballs.

CFD is the use of computers to simulate fluid flow by solving the Navier-Stokes equations. Recently, CFD has been used increasingly to visualize natural effects like water, smoke, and fire. There are a number of different CFD algorithms, but they usually involve discretizing the computational domain into a grid of cells. Stam's stable fluids approach is a very suitable CFD algorithm for applications in computer graphics, where visual results and performance are more important than accuracy.

Graphics hardware has evolved into fully programmable GPUs, capable of general-purpose computations. The programmable parts of the GPU are the vertex and fragment processors. General-purpose computation on the GPU usually only involves the fragment processor, performing computations by processing fragments, reading from input textures, and storing results in output textures.

# Chapter 3

# Previous work

In this chapter we look at previous work related to our goal of simulating and visualizing fire in real-time. We have chosen to divide the process of achieving visually convincing fire effects into two parts: simulation of fire and visualization of fire. Although these two parts can sometimes be quite intermixed, usually it is possible to make at least a certain distinction between them. The main reason for doing this division, except for achieving more clarity, is that different simulation and visualization approaches can often be combined. Thus, it makes sense to describe the two as independently as possible.

In addition to presenting previous work in simulation and visualization of fire, we look at recent work in closely related areas. First, computational fluid dynamics is the most common used method for simulating fire, thus we look at recent advances in the field. Finally, we look at recent work in utilizing the GPU to achieve efficient particle systems and volume rendering, two approaches used for visualizing fire.

## 3.1 Simulation of fire

Recently, due to both advances in research and growth in computational processing power, there has been more focus on physically based approaches for simulating fire. The advantage of physically based approaches is that they are often more flexible and less ad hoc than traditional approaches, and scale better for different types of fire effects. With a physically based approach, which to a certain level of detail approximates the physical phenomenon to be reproduced, the hope is that increased simulation accuracy is sufficient for converging to an accurate solution. Currently, the processing power offered by commodity hardware, including graphics processors, makes it possible to simulate physical processes in real-time.

Previous non-physically based approaches include particle systems [Ree83], particle chains [PP94, BPP01, Psz04], and stochastic models [SF95]. In the remainder of this section we focus on physically based approaches for simulating fire. A lot of recent approaches use fluid simulations to simulate fire [MK02, NFJ02, WLMK02, IMDN05, KW05, BLLR06] and other combustion processes like explosions [YOH00, FOA03, RNGF03, KW05]. Although there has been approaches for modeling the deformation of burning objects [MK05] or fuel consumption [ZWF$^+$03] in combination with fire simulation, due to our demand of real-time simulation and visualization and our main focus on the visual quality of the fire itself we will not focus further on them.

The rest of this section is organized as follows. First, we look at how fluid simulation is utilized to simulate the overall behavior and motion of fire. Next, we look at how the combustion process is modeled, burning fuel to create the hot gaseous products whose black-body radiation we associate with fire. Finally, we look at vorticity confinement, an approach for increasing the turbulence of the fire, attempting to create flickering and chaotic flames.

### 3.1.1 Fluid simulation

The incompressible Navier-Stokes equations, describing the dynamics of fluid flow, are used for simulating the mass and heat transfer aspects of fire in various ways by [MK02, NFJ02, IMDN05, KW05, BLLR06]. They use variations of the stable fluids approach described earlier in section 2.3.2 to solve the equations in a computational domain.

In [NFJ02], the flow of vaporized fuel and the flow of hot gaseous products are modeled independently using a separate set of Navier-Stokes equations for each. The computational domain is discretized into $N^3$ voxels with uniform spacing $h$. Solving the Navier-Stokes equations for the two fluids results in two separate velocity fields controlling the motion of respectively the vaporized fuel and hot gaseous products. As the velocity in the normal direction is discontinuous across the implicit surface separating the two velocity fields, the ghost fluid method is used when sampling across this implicit surface. Density (smoke and soot) and temperature can either be simulated in separate fields controlled by the velocity field of the hot gaseous products, or specified by animator-defined falloff-curves. These curves specify the amount of density or temperature at various times after the combustion reaction. The velocity field of the hot gaseous products is affected by the temperature as hot gases tend to rise due to buoyancy. This is modeled by an external force, proportional to the temperature, acting on the velocity field.

[MK02] use the Navier-Stokes equations to control the motion of a three-gas system consisting of oxidizing air, fuel gases, and exhaust gases. The spread of heat is also modeled. As in [NFJ02], the computational domain is discretized into $N^3$ voxels, and solving the Navier-Stokes equations results in a velocity field controlling the motion of air, which acts as the fluid in the simulation. The fuel gases, exhaust gases, and heat are simulated explicitly in separate fields, and advected by the velocity field of the air. In addition, the three fields are affected by dissipation and the temperature field is affected by diffusion simulating the spread of heat. The fuel gases, exhaust gases, and heat in turn affect the velocity field in terms of a gravity force and a buoyancy force. The gravity force is proportional to the amount of fuel gases and exhaust gases while the buoyancy force is proportional to the temperature. The buoyancy force is the most crucial for obtaining the correct flame shape.

A velocity field is also evolved in [IMDN05] using the Navier-Stokes equations. The velocity field is defined in a 3D voxel grid and is used to advect a temperature field. In addition, the temperature field is affected by diffusion as in [MK02]. The velocity field is also here affected by a buoyancy force proportional to the temperature. A fuel field is also included in the simulation, but it is not affected by the velocity field. The temperature field is used to compute the heat conducted in the fuel and to trigger the combustion of fuel. Note that they do not simulate the motion of exhaust gases.

In [BLLR06], the Navier-Stokes equations are used to simulate the dynamics of simple flames produced by candles and oil lamps. The surface of the flame model is represented by a few particle chains, which in turn are used as control points for a NURBS surface. The velocity field from the fluid simulation, defined in a 3D voxel grid, is used to advect the particles belonging to the particle chains. Fuel and exhaust gases and temperature are not explicitly modeled, but they approximate a buoyancy force by adding a vertical force linearly decreasing with height to each cell of the discretized velocity field. They also introduce wind-like effects by adding external forces around the flame.

In [KW05], the Navier-Stokes equations are solved using the stable fluids approach on the GPU, modeling the fluid flow in which the fire is located. The velocity field resulting from the fluid simulation is used to transport heat that is constantly injected, and finally the velocity field is used to advect fire particles in the visualization step. When simulating fire, the velocity field is affected by a buoyancy force and pre-computed pressure templates that are scaled by the temperature. Fuel and exhaust gases are not a part of the fluid simulation. The fluid simulation is performed in 2D as opposed to [MK02], [NFJ02], [IMDN05], and [BLLR06] who perform the it in 3D. To achieve a 3D flame from several individual 2D simulations volumetric extrusion from [RNGF03] is used.

A different approach for fluid simulation is used by [WLMK02]. Instead of the Navier-Stokes equations, the Lattice Boltzmann Model (LBM) is used to simulate the evolution of the fire. Utilizing the LBM results in a 3D velocity field which is used in the visualization step to advect fire display primitives. The velocity field is affected by wind direction and non-burning objects, and to model the generation of smoke a temperature field is used. Although they achieve very fast simulations by implementing the LBM on the GPU, it is unclear how accurate it is compared to stable fluids, and thus we do not focus further on it.

### 3.1.2 Combustion modeling

There have been several approaches to modeling the combustion which occurs when sufficient heat causes oxidizer to react with fuel gas, creating more heat, exhaust gas, and other byproducts. [NFJ02] implicitly model the combustion by simulating the flame's blue core. The surface of the blue core approximates the reaction zone separating the fuel gas and exhaust gas, and is modeled using the level set method. The ghost fluid method is used in the fluid simulation when sampling across the blue core surface to simulate the expansion that occurs when fuel gas is converted to exhaust gas in the combustion reaction.

Another physically based approach to combustion modeling is done by [MK02]. They use an explicit combustion model by simulating the combustion process in each grid cell in the computational domain. Based on the amount of fuel gas and oxygen in a grid cell a combustion parameter is computed. The fuel gas, exhaust gas, and temperature values are then updated based on the combustion parameter if the temperature in the grid cell is above the ignition temperature of the fuel gas. Different combustion reactions can be modeled by varying a burning rate parameter, governing the percentage of the fuel gas that can be burned in a second, and a stoichiometric mixture parameter, governing the amount of oxygen required to burn one unit of fuel gas.

[IMDN05] also model the combustion process explicitly, but they only consider the decrease in concentration of fuel and not the generation of byproducts. The rate of the reaction is among other affected by a reaction frequency factor and the temperature. They assume that there always exists enough oxygen to react with the fuel.

The combustion approaches used by [WLMK02] and [KW05] are less physically based and more ad hoc. [WLMK02] model combustion indirectly by injecting fire display primitives at inlets and advecting them using the velocity field from the LBM. Each fire display primitive contains a certain amount of fuel, and when the fuel is consumed the fire display primitive is removed from the simulation.

[KW05] also model fire display primitives, or particles, advected by a velocity field, however without directly simulating the combustion process. Pressure templates scaled by temperature are instead used to disturb the flow at the base of the fire, causing turbulent effects as the flames rise.

### 3.1.3 Vorticity confinement

One of the disadvantages of Stam's stable fluid approach is its inherently high amount of numerical dissipation. To counter this, [FSJ01] combined the stable fluid solver with an artificial vorticity confinement force which increases the turbulent features of the velocity field. The vorticity confinement method finds rotational movement in each of the velocity field grid cells by looking at adjacent grid cells, and computes a vorticity force which enhances or maintains the rotational movement, thus reducing the amount of rotational movement lost due to numerical dissipation. In fact, vorticity confinement can be used to create even more turbulence in the velocity field. Vorticity confinement has been used in combination with fire simulation by [NFJ02], [IMDN05], and [KW05], attempting to create more turbulent and chaotic flames.

## 3.2 Visualization of fire

The results from a fire simulation can be visualized in a lot of different ways. In this section we focus on the two main current approaches for visualizing fire in real-time, volume rendering and particle systems respectively. In addition, we briefly mention offline approaches that are too computationally expensive for real-time use. Finally, we look at methods for dynamic illumination of the scene surrounding the fire.

### 3.2.1 Volume rendering

A hardware based volume rendering method is used in [MK02] for fire visualization, as shown in figure 3.1. The output of the fire simulation is voxelized data of the fuel gas, exhaust gas, and heat in the computational domain, and each voxel is replaced by a semitransparent polygon where the level of transparency is controlled by the density of fuel gas and exhaust gas in the voxel. The fuel gas is shown in yellow and the reaction zone where the combustion occurs is shown in red. The amount of fuel combusting in a voxel is used to decide the intensity of the color red. The exhaust gas is just shown as smoke and it is almost transparent to avoid obscuring the flame.

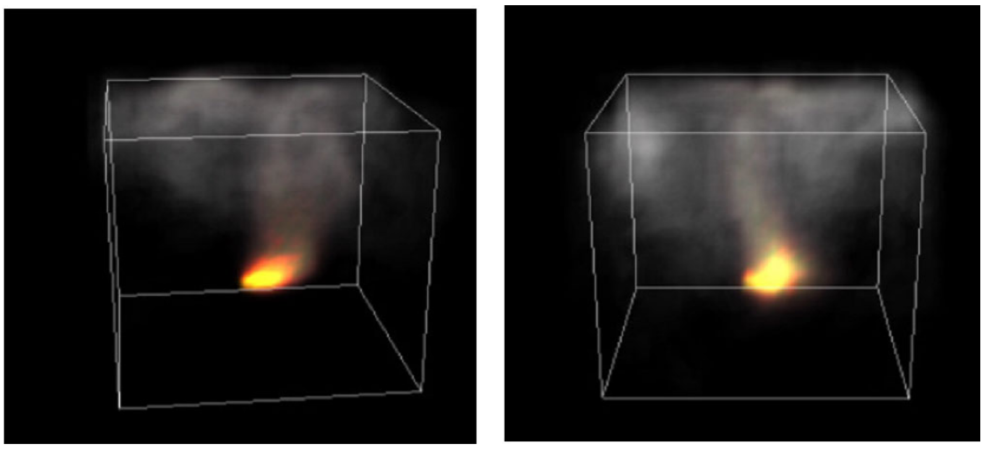

Figure 3.1: Two timesteps of a visualization of a flame in a closed environment [MK02].

[MK02] make it clear that the visual quality was of secondary concern, the focus instead being on creating an interactive simulation with a realistically behaving fire. The visualization method is slightly inaccurate, as the fuel gas itself is not a part of the visual phenomena of fire. The yellowish-orange color that is associated with fire comes from the radiance emitted by the exhaust gas at high temperature and the smoke and soot is just exhaust gas that has cooled down.

A similar volume rendering method is used by [IMDN05]. Instead of replacing each voxel by a semi-transparent polygon they place a texture at the center of each voxel and render the textures using graphics hardware. The color of the textures is calculated using the temperature distribution from the simulation and Planck's formula for black-body radiation. Figure 3.2 shows a fire visualized using this method.

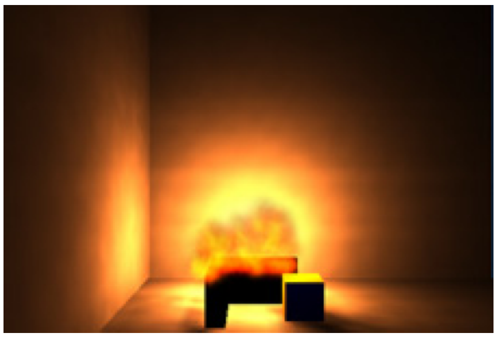

Figure 3.2: Room illuminated by a volume rendered fire [IMDN05].

[KCR99] use a volume rendering approach to visualize animated amorphous materials such as fire, smoke, and dust. A fire effect is modeled with a coarse, regular voxel grid of opacity values, representing the overall shape of the fire. The voxel grid is static as no underlying physically-based method is used to create object dynamics. To visualize the volume a set of textures are selected and each texture is weighted with an anisotropic footprint function, producing textured splats with embedded details. These fine scale details compensate for the coarse voxel grid used. A splatting technique is then used to render each voxel independently as a textured splat in back-to-front order with respect to the viewport. The color of a voxel is generated from a color lookup table, while the textured splat determines the opacity. To create local dynamics and the illusion of motion the set of textures are cycled in each voxel. Figure 3.3 shows an example scene where fire is volume rendered.

### 3.2.2 Particle systems

[KW05] use a GPU implemented particle system to visualize the fire. For each particle a velocity vector is reconstructed at the particle's current position using 2D velocity field slices from the fluid simulation and volumetric extrusion as presented by [RNGF03], which we describe further in section 3.4.2. This velocity vector is then used to move the particle an Euler step in the direction of the flow, and the new particle position is stored. Finally, the particles are rendered as point sprites, where the sprite color is modified according to a specified color table or using flow quantities like density or temperature. In the upper part of the simulation where the temperature decreases, the

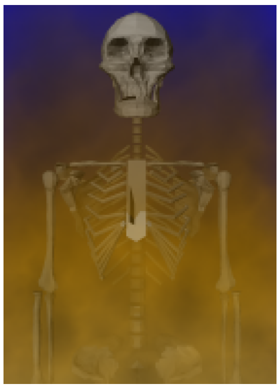
Figure 3.3: A 3D model inside a fire volume [KCR99].

fire turns into smoke and the sprite color is modified accordingly. Because the particles are rendered as rather small point sprites, a large number of particles are needed in the visualization. The particle system used is similar to the ones presented later in section 3.5, where we discuss general particle system implementation on the GPU. Figure 3.4 shows a campfire with smoke, simulated using 2D fluid simulations with volumetric extrusion and rendered using a particle system with textured point sprites.

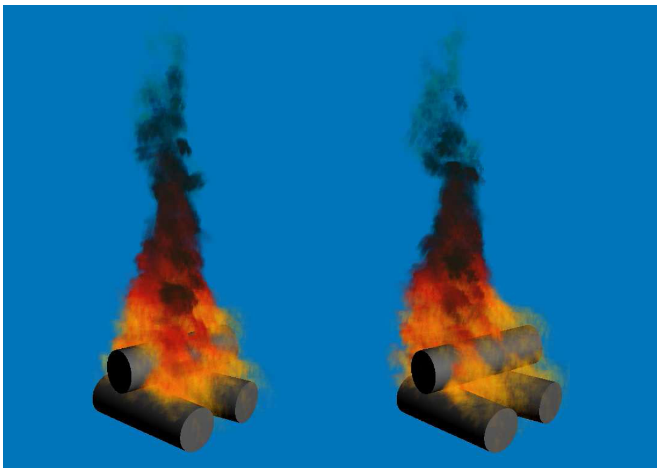
Figure 3.4: Two timesteps of a campfire visualization [KW05].

In [WLMK02], texture splats are used to visualize the fire. The result of the LBM algorithm is a 3D velocity volume with a velocity vector defined at each lattice cell, and this velocity volume is used to advect the display primitives (fire particles) that are inserted into the computational domain. For each fire particle the velocity vector at the particle's current position is computed using trilinear interpolation from the velocity volume, and the interpolated velocity vector is used to move the fire particle to its new position. The fire particle is removed from the system and reinserted if it moves out of the computational domain or if the fuel gas it carries is consumed due to combustion. The color and transparency of the fire particle depend on where it is located in the fire. A black-body radiation color table is used to determine the colors at different locations in the fire based on the temperature, and the higher the temperature the brighter the color. To include smoke in the visualization, the temperature of each display primitive is tracked, and when it decreases to a certain point the fire particle changes to a smoke particle. A different color table is used to assign colors to the smoke particles. Finally, the particles are rendered back-to-front assuming the light is scattered toward the viewer. Each particle is associated with a texture splat, a small image of turbulent details, in order to add fine-scale details to the fire. By using texture splats instead of point particles as rendering primitives, the computational complexity of a large particle system is avoided. In figure 3.5, a simple campfire with smoke using around 100 texture splats can be seen.

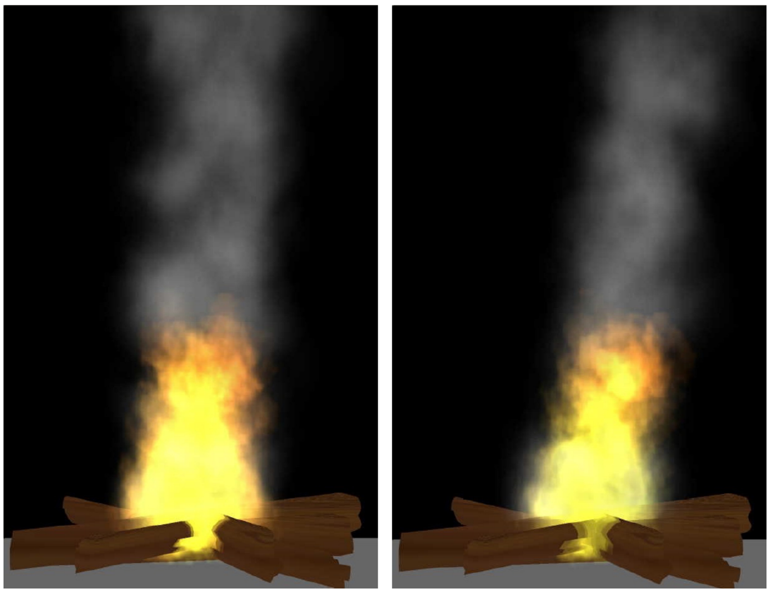

Figure 3.5: Two timesteps of a visualization of a campfire with smoke [WLMK02].

### 3.2.3 Offline approaches

[NFJ02] use a Monte Carlo ray tracing approach to visualize the hot exhaust gas produced in the simulation. They model fire and low-albedo smoke as a scattering and absorbing participating medium. The emitted spectral radiance in the fire is computed using Planck's formula, based on the fire's temperature. Finally, they model the chromatic adaption of the eyes to the color of the fire using a von Kries transformation. In figure 3.6, a very realistic offline rendering of a campfire can be seen.

In [LF02], a system for animating flames is described. The focus is on creating believable flames with total artistic and behavioral control. They do not use a numerical simulation to guide the visualization, as they claim that numerical simulations have a number of undesirable properties: they scale poorly, it is difficult for developers

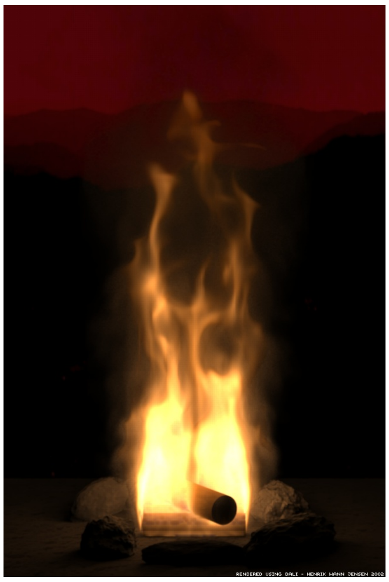

Figure 3.6: A campfire with two burning logs [NFJ02].

to place control functions on top of an underlying physical system, and it is hard to achieve the desired visual result by altering the physical parameters alone. They do not explicitly calculate the colors of the fire, instead finding a reference photograph of the flames they want to model and mapping the picture onto a two-dimensional flame shape profile. They indirectly volume render the fire volume using a large amount of particles, resulting in a sampling rate of around ten particles per pixel. Figure 3.7 shows clips from the animation movie *Shrek*. The very realistic fire breathed by the dragon is generated using the large-scale fire animation system presented in the article.

### 3.2.4 Dynamic illumination

In addition to the visual appearance of the fire itself, the effect of the fire on the surrounding environment can also be taken into account. The illumination of nearby objects caused by the fire has been included in several previous approaches. [NFJ02] achieve a full global illumination solution due to their Monte Carlo ray tracing approach. Full global illumination is currently not feasible in real-time, but even local illumination due to the fire and other light sources can enhance the realism of a scene. We will use the term dynamic illumination to mean local or global illumination of surrounding objects which varies and flickers due to the temporal and spatial variations in the fire.

[IMDN05] place point light sources at the center of each voxel of the simulation grid in order to illuminate the scene. They compute the light intensities from the simulation's temperature distribution by using Planck's formula for black-body radiation. Their approach produces realistic dynamic illumination, as shown earlier in figure 3.2. Due

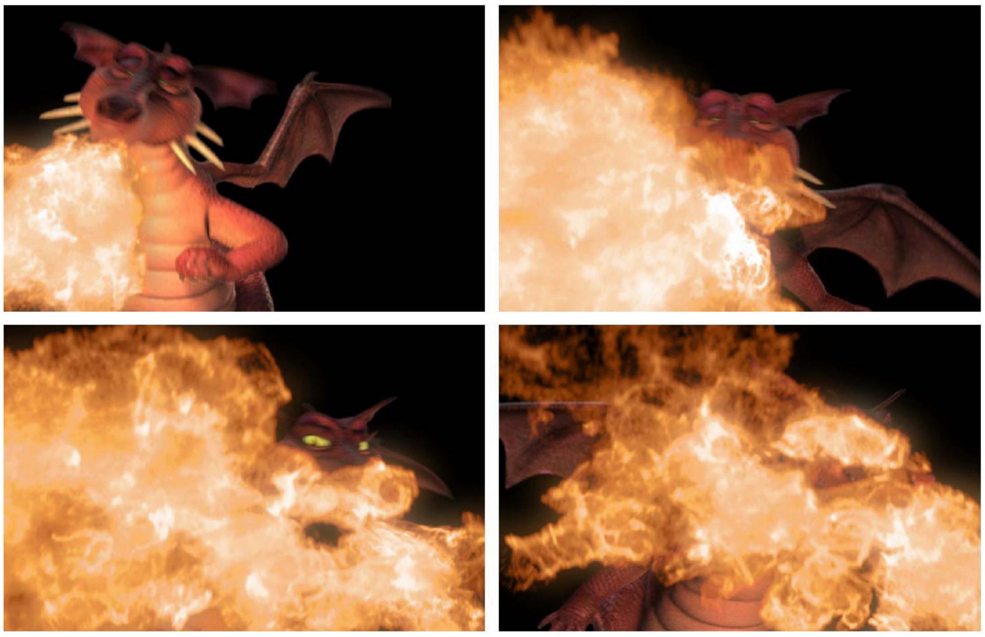

Figure 3.7: Fire breathed by the dragon from the movie *Shrek* [LF02].

to the high number of light sources their rendering times are quite high, one second for the scene shown in the figure, which uses a simulation grid of 32x32x32 voxels.

[BLLR06] achieve dynamic illumination for candle and oil lamp flames in real-time by approximating the illumination due to the flame with a photometric solid. The spectral and photometric distributions are captured from real candle and oil lamp flames by a spectrophotometer. From these distributions a photometric solid is created, defining luminous intensities for a set of zenithal and azimuthal directions. In contrast to simple point lights, the photometric solid allows non-uniform illumination distributions. The intensities of the photometric solid are stored in a 2D texture, which is used by a fragment program when rendering the surrounding environment of the flame. The dynamic illumination caused by deformations of the flame shape is approximated by applying a rotation to this texture. In addition, an attenuation factor is calculated based on the distance of a fragment from the flame and the ratio of the flame's current size to its size at rest. The final results are quite convincing, as can be seen in figure 3.8.

## 3.3 Approach comparison

In this section we compare the approaches for simulating and visualizing fire previously described in this chapter. For both the fire simulation and the visualization we have defined a set of aspects that one or more of the described approaches include or fulfill. In addition, we have defined *Real-time* as an aspect that covers both the simulation

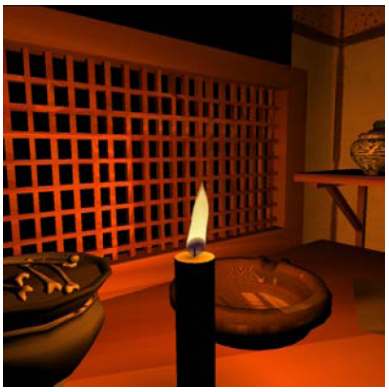

Figure 3.8: Candle flame illuminating a scene [BLLR06].

and the visualization. We consider the approach as real-time if it achieves a frame rate of 30 FPS or better for the simulation and visualization combined[1], which is the requirement we use for real-time when later evaluating our results. It is also worth pointing out that the aspect *GPU implemented* is only concerned with whether the fire simulation is implemented on the GPU and not with the visualization of the fire.

In table 3.9, the various approaches are shown in the leftmost column. An $X$ indicates that a given approach includes or fulfills the given aspect. As can be seen, most approaches have a fire simulation that to a certain degree is physically based. Nevertheless, [MK02], [NFJ02], and [IMDN05] can be picked as the ones with the most physically correct fire simulation.

## 3.4 Real-time computational fluid dynamics

As seen from section 3.1, fluid simulations have been used successfully to simulate fire in real-time. In this section we look at recent advances in real-time CFD. Our focus is on GPU implementations of fluid simulations, as these are an order of magnitude faster than the CPU counterparts.

There have been several important developments in the field of real-time computational fluid dynamics lately. The influential approach presented by Jos Stam in [Sta99] has recently been adapted for processing on the GPU [Har04, LLW04, WLL04, KW05]. This section focuses on various approaches for implementing Stam's stable fluids approach on the GPU.

[1][MK02] achieved a frame rate of around 20 FPS and we consider their approach real-time because it would most likely achieve more than 30 FPS on today's hardware.

| Approach | | Simulation | | | | | | | | | Visualization | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Real-time | Stable fluids | GPU implemented | Temperature field | Fuel gas field | Exhaust gas field | Physically based combustion model | Buoyancy force | Vorticity confinement | Volumetric extrusion | Black-body radiation | Blue core | Particle system | Volume rendering | Smoke | Dynamic illumination |
| [KCR99] | X | | | | | | | | | | | | | X | X | |
| [MK02] | X | X | | X | X | X | X | X | | | | | | X | X | |
| [NFJ02] | | X | | X | X | X | X | X | X | | X | X | | | X | X |
| [WLMK02] | X | | X | X | | | | | | | X | | X | | X | |
| [IMDN05] | | X | | X | X | | X | X | X | | X | | | X | | X |
| [KW05] | X | X | X | X | | | | X | X | X | | | X | | X | |
| [BLLR06] | X | X | | | | | | X | | | | X | | | | X |

Figure 3.9: Comparison of different approaches.

### 3.4.1 Fluid simulation on the GPU

Mapping the stable fluid solver to the GPU helps achieve real-time results and allows simulation of higher resolution grids. The mapping is done by storing data in textures and performing computations in fragment programs, as described earlier in section 2.4. In [WLL04] a stable solver was implemented in 2D on both the CPU and GPU. The speedup factor between the CPU and GPU varied from 11.1 to 14.7 for large grid sizes, on a Pentium 2.8 GHz processor with 2 GB main memory and a GeForce FX5950 Ultra with 256 MB video memory and 375 MHz core frequency. These results show that offloading the task of simulating the fluids from the CPU to the GPU can result in a large performance increase.

In broad terms, the method used by [WLL04] can be summarized in the following steps:

1. The force terms are computed and added to the current velocity field.
2. The advection term is computed and added to the velocity field.
3. Jacobi iteration, with one rendering pass per iteration, is used to add the diffusion term to the viscous velocity field.
4. The divergence of the current velocity field is calculated.

5. The Poisson equation is computed and solved by Jacobi iteration like in step 3, yielding a pressure distribution field.

6. The velocity field is updated by subtracting the gradient of the pressure distribution calculated in step 5.

In addition to these steps they also handled arbitrary boundary conditions, allowing them to insert obstacles into the fluid domain. The fluid is blocked by and flows around the obstacles, causing interesting effects as shown in figure 3.10.

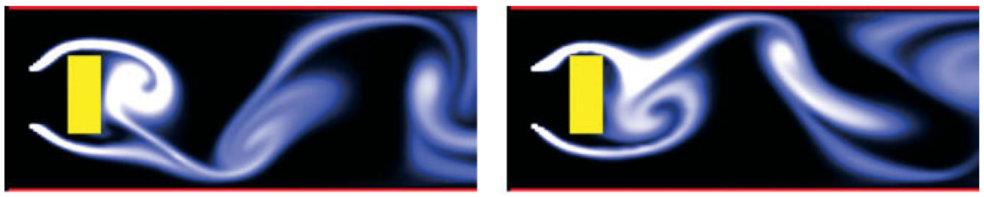
Figure 3.10: A density flowing around an obstacle [WLL04].

[WLL04] use the fluid solver to evolve a velocity field as well as density and temperature fields. A buoyancy force is calculated from density and temperature to simulate their effect on the velocity field. In addition, the vorticity confinement method is used to reintroduce small-scale turbulence lost due to numerical dissipation. Instead of repeating the steps described above for the x and y components of the velocity field, as well as for the density and temperature fields, all the field values are packed into the four RGBA-channels of a single floating-point texture.

In [LLW04] the solver is extended to the 3D domain. This extension could be achieved by slicing the 3D domain into 2D textures along an arbitrary axis. A 64x64x64 grid could thus be represented by 64 textures, each having the dimensions 64x64. However, each computational step would have to be performed once for each slice, which would get quite expensive at large grid sizes. Instead, the 2D texture slices are packed into one single 2D texture by tiling them. The resulting 2D texture, containing 2D slices of the 3D computational domain, is called a flat 3D texture. For example, a 32x32x32 grid can be tiled into a 256x128 2D texture. As some of the computational steps require access to grid neighbors, another 2D texture contains the grid cell indices of neighboring slices. The 3D extension also requires the use of two floating-point textures to store the field values, one for the x, y, and z components of the velocity field, and one for the density and temperature.

[HBSL03] also implement a 3D fluid solver on the GPU in order to simulate cloud dynamics. Like [LLW04] they use flat 3D textures, and they optimize the calculation of the Poisson equation by using a vectorized Jacobi iteration. In order to achieve real-time simulation they perform the fluid solver computation over several frames, effectively amortizing the simulation cost.

### 3.4.2 Volumetric extrusion

A 3D fluid simulation implemented on the GPU is capable of real-time results using rather small simulation grids, as shown in [LLW04]. However, on very large grids a 3D simulation is impractical. In [RNGF03] they propose volumetric extrusion for deriving a 3D velocity field from several 2D velocity fields stemming from independent 2D fluid simulations, where the derived velocity field is similar to a velocity field obtained from a 3D fluid simulation. This method is much faster than a full 3D simulation, as a few 2D fluid simulations require far less computation time. The 2D velocity fields are seen as 2D cross-sections of a flow field in 3D space, and interpolation methods to fill in the empty regions between the cross-sections are defined. One possibility is to tile the cross-sections in a cylindrical fashion, as seen in figure 3.11, and perform cylindrical interpolation along an arc of constant radius from the vertical axis between the two cross-sections. If the phenomena have approximate axial symmetry, like bonfires and torches, this interpolation variant is especially suitable.

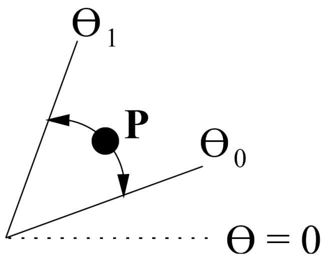

Figure 3.11: Cylindrical interpolation (top view) [RNGF03].

[KW05] use the volumetric extrusion method from [RNGF03] in combination with a 2D fluid simulation to model volumetric effects like smoke, fire, and explosions. The 3D domain is represented by a cylinder, and a set of 2D slices from the cylinder are simulated independently. To retrieve field values throughout the 3D domain, cylindrical linear interpolation is used, fetching values from the two closest slices and combining them. In addition, a stochastic fractal distribution is used to perturb the 2D slices when interpolating, in order to introduce turbulent small-scale features. The method is much faster than performing a full 3D simulation, but it limits the number of fluid phenomena that can be captured and is less accurate.

## 3.5 Particle systems on the GPU

Particle systems can be used to model fuzzy objects such as fire and smoke, which have irregular, complex, and ill defined surfaces [Ree83]. A particle system is a collection of small particles that together represent a fuzzy object. Particles are generated into a system, move and change from within the system, and die from the system. Figure 3.12 shows an early use of particle systems in the movie *Star Trek II: The Wrath of Khan.*

A wall of fire spreading over the surface of a planet is visualized using a distribution of particle systems.

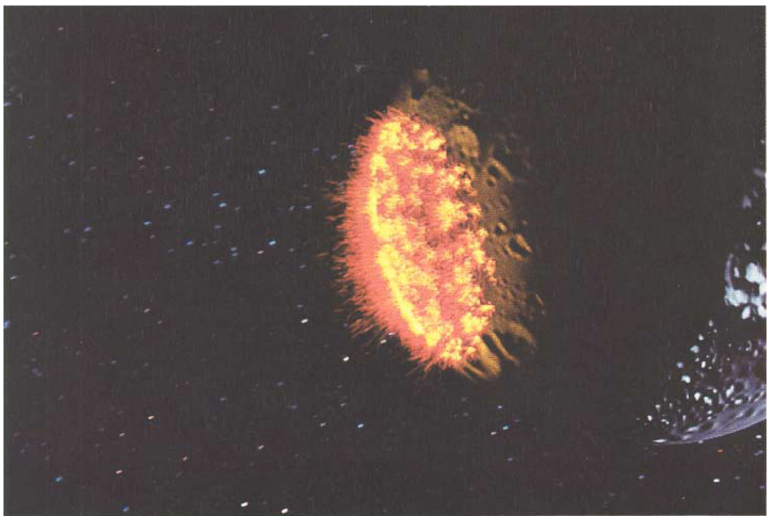

Figure 3.12: Particle systems used in *Star Trek II: The Wrath of Khan* [Ree83].

Recently, modern GPUs have become capable of performing both simulation and rendering of particle systems [KSW04, Lat04], due to the addition of new render to vertex buffer and vertex texture fetch functionality. In this section we give an overview over the simulation and rendering of particles on the GPU.

### 3.5.1 Representation

[KSW04] and [Lat04] use textures on the GPU to represent particles, in order to utilize the fragment processors to perform the particle simulation. Although the textures are conceptually treated as one-dimensional arrays, they are actually two-dimensional due to hardware size restrictions. Particle positions and velocities are stored in separate textures, treating the RGB color components of each texel as x, y, and z coordinates. As textures can not be used for input and output at the same time, a pair of textures are used in combination with a double buffering technique for both positions and velocities. In addition, [Lat04] store static data that is not dynamically updated in textures which do not use the double buffering approach. This static data can include particle time of birth, particle type, and more.

### 3.5.2 Particle simulation

The particle simulation consists of updating particle positions and velocities, as well as processing the birth and death of particles. In [Lat04], the CPU is used to process particle births by determining an available index in the particle textures. Similarly, the CPU reclaims the freed index of dying particles. The GPU sets the positions of dead particles to an invisible area, e.g. infinity, which increases rendering performance.

Particle positions and velocities are updated in two separate rendering passes. Local or global forces can be used when updating the particle velocities, whereas the particle positions are updated solely based on the corresponding velocities. Both velocities and positions can be updated using Euler integration.

### 3.5.3 Particle rendering

Rendering the particles requires transferring the position data stored in textures to vertices. In [Lat04], two approaches are suggested: vertex textures and über-buffers. Vertex textures are used in the vertex program to set vertex positions based on a texture fetch from the particle position texture. Particles are rendered with point sprites, triangles, or quads by drawing a static vertex buffer of respectively one, three, or four vertices per particle. The other approach, über-buffers, basically copies pixel data from textures directly to vertex data using an OpenGL extension.

If alpha blending is used and correct blending is desired, the particles might need to be sorted before rendering. This is done in both [KSW04] and [Lat04]. The latter however spread the sorting over several frames, avoiding the large cost of sorting all particles each frame.

## 3.6 Volume rendering on the GPU

[DCH88] introduced volume rendering as a method for rendering images of volumes containing mixtures of materials. A light model is used to model the contribution of a voxel on a light ray traveling through it. Materials may act as translucent filters, absorbing incoming light, or may be luminous and emit outgoing light. To render the volume they first transform the volume to the viewing pyramid, and then overlay the colored planes of the volume from back to front. A CT scan of a sea otter volume rendered using their approach is shown in figure 3.13.

[KW05] use volume rendering to visualize explosions and other volumetric effects based on underlying fluid simulations. Fire has been volume rendered by among other [KCR99], [MK02], and [IMDN05]. Although volume rendering has traditionally been to slow for use in real-time applications, the method is capable of high quality visualization and is therefore an interesting option for visualizing fire. Utilizing the GPU has now become the best approach to achieve real-time volume rendering of dynamic data sets.

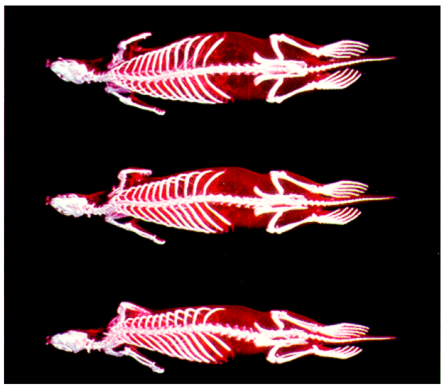

Figure 3.13: Volume rendering of a CT scan of a sea otter [DCH88].

### 3.6.1 Ray marching

Using ray marching for volume rendering is very suitable for GPU implementation, as done by [KW03]. Fragment programs are used in [KW03] to cast rays through the volume, each fragment representing a single ray. As they use Pixel Shader 2.0 functionality, which does not support dynamic looping, they perform a constant number of ray steps through the volume in each of several rendering passes. At each ray step, the ray position is used to sample a scalar value from the data volume. This scalar value is then often used in combination with a color table to lookup a color value, which is finally blended in some way with the accumulated color value.

To compute the rays used to step through the volume, [KW03] simply render the front and the back faces of the volume data set's bounding box. When the front faces are rendered, a fragment program stores the ray entry points in a texture. Next, this texture is used by another fragment program when rendering the back faces of the bounding box, in order to compute the direction and length of each of the rays.

### 3.6.2 Acceleration techniques

To speed up the volumetric ray-casting, [KW03] use early ray termination and empty-space skipping. Early ray termination is performed between each of the main rendering passes, checking if threshold values for opacity have been reached and writing to the z-buffer to avoid further fragment operations along the same ray. Thus, this acceleration technique is only applicable when using several passes as opposed to dynamic looping. Empty-space skipping involves another intermediate pass and requires precalculating a 3D texture map, specifying at each voxel the minimum and maximum scalar values contained in a block of size $8^3$ within the volume data set. These minimum and maximum values are then used to be able to skip entire blocks if the values contained within would not affect the resulting color value. As with early ray termination, this technique can not be used with dynamic looping, and in addition it requires that the data set is static. As a fire volume renderer needs to visualize a dynamic data set (the results from a fire simulation), empty-space skipping would not be useful.

## 3.7 Summary

We have presented physically based approaches for simulating and visualizing fire. Common to most physically based approaches is the use of a fluid simulation to perform the fire simulation. The fluid simulation is used to evolve a velocity field in combination with fuel gas, exhaust gas, and temperature fields. Different approaches choose to simulate some or all of these fields. Next, an explicit or implicit combustion model is used to simulate the burning of fuel gas and the generation of exhaust gas and heat. Usually, a buoyancy force affects the velocity field, making exhaust gas rise due to heat. Vorticity confinement can be used to increase the turbulence of a velocity field. Fire simulations using an underlying fluid simulation have been implemented both on the CPU and on the GPU.

Volume rendering and particle systems are the most common approaches for visualizing fire in real-time. Volume rendering has been done in various ways by visualizing the voxels of the fire simulation based on the fire simulation fields. The particle system approaches we have described involve moving the particles based on the fire's velocity field and choosing particle colors based on the temperature and exhaust gas fields. The particles can be textured to create more low-level detail. We have also presented dynamic illumination, which can be used to model the illumination of other objects in the scene by the fire. The illumination caused by the fire has previously been approximated by point lights or photometric solids.

Stable fluids, the most common approach for performing fluid simulations for real-time fire simulations, is especially suited for implementation on the GPU. A GPU stable fluid solver is usually an order of magnitude faster than its CPU counter-part. On the GPU the fire simulation fields are represented as textures, and operations are performed by fragment programs. Even on the GPU it is expensive to perform fluid simulations using large 3D grids. Instead, volumetric extrusion can be used to combine a set of 2D simulations into a 3D simulation by cylindrical interpolation. Volumetric extrusion can therefore be beneficial for simulating phenomena with axial symmetry, like bonfires and torches.

Finally, we have looked at the implementation of particle systems and volume rendering on the GPU. Particle systems are especially suitable for implementation on the GPU. Particle positions and velocities are stored in textures and updated by fragment programs. Particles can be rendered directly on the GPU by fetching positions from the particle position texture in a vertex program using Vertex Shader 3.0 functionality. Volume rendering can also be performed efficiently on the GPU if the volume data is stored in textures. The volume rendering can be done by ray marching in a fragment program, using Pixel Shader 3.0 functionality supporting dynamic loops.

# Chapter 4

# Simulation of fire

In this chapter we present our physically based approach for simulating fire, whereas we in the next chapter present our visualization of the results from the simulation. Our fire simulation is largely based on the approach presented in [MK02], which combines the stable fluid solver from [Sta99] with a combustion process modeling fuel gas, exhaust gas, and temperature fields. The fields are limited to a finite computational domain, which represents the area where the fire is located, and are discretized into a grid or voxel structure. In addition, we use the vorticity confinement method used in [NFJ02], [IMDN05], and [KW05] to create a more turbulent fire. We have also adopted the compressible fluid approximation used by [FOA03], attempting to model the expansion that occurs when fuel gas combusts. In order to increase computational efficiency when simulating rotationally symmetric flames, we adopt volumetric extrusion as presented by [RNGF03] and also used by [KW05].

In the rest of this chapter we first give a general overview of our method, before giving a detailed presentation of the simulation fields and equations governing them. Next, we briefly discuss the boundary conditions of the simulation. We then discuss how to sample from the simulation fields, using plain interpolation or volumetric extrusion. Finally, we present the complete simulation algorithm.

## 4.1 Overview

The fire simulation models the burning of fuel gas in a computational domain. We simulate the evolution of a fuel gas field, an exhaust gas field, and a temperature field, in co-evolution with a velocity field. We will refer to the fuel gas, exhaust gas, and temperature fields collectively as the fire density fields. The fire density fields are scalar fields representing the amount of fuel gas, exhaust gas, and heat respectively, whereas the velocity field is a vector field representing the direction and speed of the fluid, the air in which the fire density fields flow. A set of differential equations govern the evolution of the fields, which are discretized into a grid or voxel structure representing a computational domain in respectively 2D or 3D. The equations are solved on a cell by cell basis.

The main process of the fire simulation is the fire combustion, which converts fuel gas to exhaust gas and heat when the temperature exceeds the fuel's ignition threshold. Buoyancy due to heat affects the velocity field, causing the hot exhaust gas to rise, which in combination with vorticity confinement is responsible for the characteristic fire-like motion. The fire simulation is also extended to model the mass expansion occurring during the combustion reaction.

### 4.1.1 Computational domain

When the simulation is performed in 2D, a grid structure is used to represent the discretized fields, as shown in figure 4.1. There are two kinds of cells: interior cells and boundary cells. Each cell contains a corresponding field value. Both density and velocity values are defined at the center of the cells of the computational domain. Defining all values at cell centers is known as collocated grid as opposed to staggered grid, where velocities are defined at cell sides. Collocated grids allow for more straightforward implementation than staggered grids [Sta99].

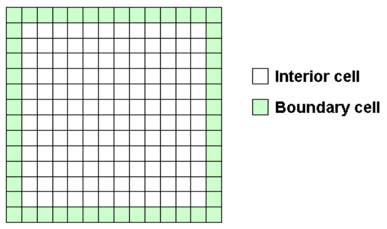

Figure 4.1: 2D computational domain.

Similarly, for 3D simulation a voxel structure represents the discretized fields. A cross-section of a voxel domain is shown in figure 4.2. As for 2D grids, the voxel domain has interior voxels and boundary voxels, and contains field values defined at voxel centers.

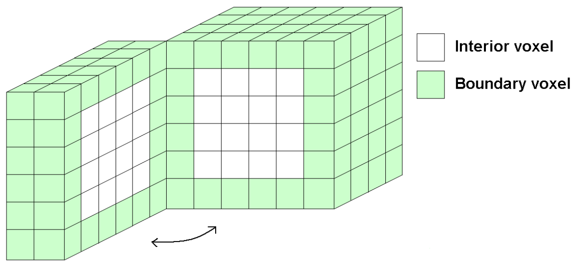

Figure 4.2: Cross-section of a 3D computational domain.

The equations presented later in this chapter are all solved on a local per cell basis. Sampling from neighboring cells or voxels is required when solving several of the equations, which is why we need special treatment for the boundary values. Exactly how the boundary values are set depends on which boundary condition is used. The boundary conditions are explained in more detail later in section 4.3.

### 4.1.2 Simulation overview

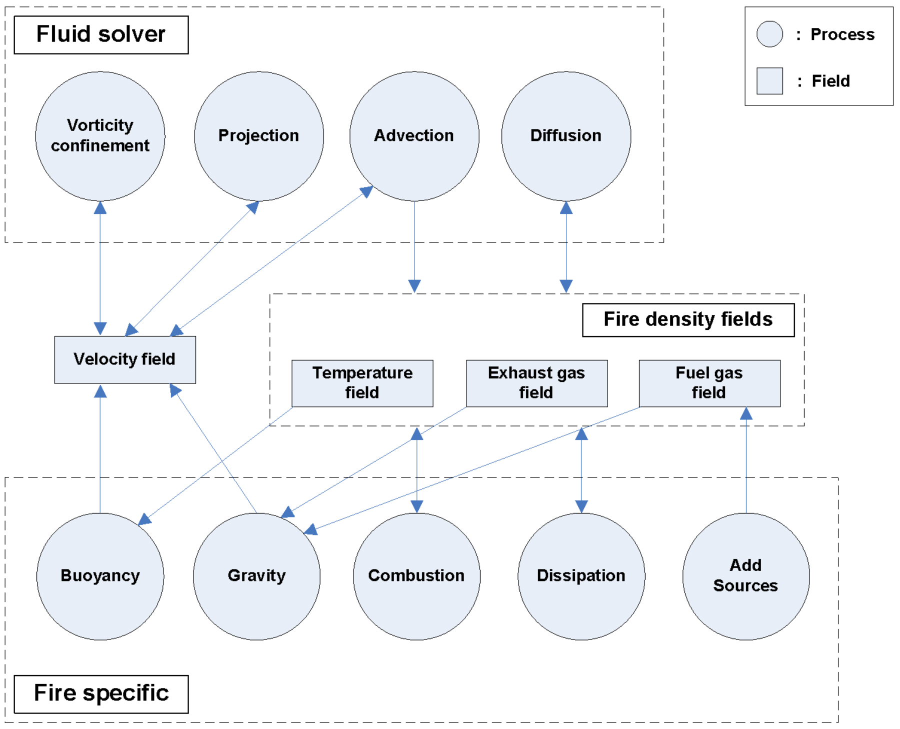

Figure 4.3: Simulation processes and fields.

Figure 4.3 shows the essential processes involved in the fire simulation. The boxes contain the important fields, and the circles show the main processes. Arrows indicate the relationship between fields and processes; an arrow from a field to a process means that the field is used as input for that process, whereas an arrow going from a process to a field means that the field is updated as a result of the process. The fire specific processes and the *Vorticity confinement* process all result in forces which affect the velocity or fire density fields. A force in this context is anything which adds or subtracts quantities from the fields.

The simulation consists of several steps, each consisting of a number of processes. First, we calculate the forces acting on the velocity field, represented by the following processes:

- *Vorticity confinement* - locates and enhances turbulent features in the velocity field.

- *Gravity* - models how fuel and exhaust gases are pulled down due to gravity.
- *Buoyancy* - models air rising due to heat, based on the temperature field.

Next, we calculate the fire density field forces, represented by the following processes:

- *Combustion* - models the reaction that happens when fuel gas reacts with oxygen due to heat, creating exhaust gas and more heat.
- *Dissipation* - models the dissipation or dispersal of exhaust gas, fuel gas, and temperature fields, thus limiting the range of the fire.
- *Add sources* - keeps the fire alive by injecting fuel gas.

The fluid solver is then used to simulate the behavior of the velocity field, as illustrated by the following processes:

- *Advection of velocity* - self-advects the velocity field, moving the velocity along itself.
- *Projection* - ensures that the velocity field is mass conserving.

Projection in combination with self-advection is a critical part of achieving fluid-like motion.

Next, the fluid solver uses the velocity field to simulate the behavior of the fire density fields, as shown by the following processes:

- *Advection of fire densities* - moves the fuel gas, exhaust gas, and heat according to the velocity field.
- *Diffusion* - models the natural tendency of gases and heat to spread.

## 4.2 Simulation fields and equations

In this section we give a detailed presentation of the velocity and fire density fields and the equations governing them. Although we present the equations for a 3D simulation, corresponding equations are used in the 2D case.

The velocity field is modeled using the Navier-Stokes equations and influenced by vorticity confinement, gravity, and buoyancy. Optionally, mass expansion can be used to simulate the expansion that occurs during combustion. For clarity, mass expansion is presented separately from the main velocity field equations. The fire density fields are set to follow the velocity field, and are in addition affected by combustion, diffusion, dissipation, and user controlled sources.

### 4.2.1 Velocity field

The velocity field **u** is a vector field specifying the direction and speed that air and gas move in. The velocity field is governed by the Navier-Stokes equations for incompressible flow with zero viscosity, as shown in the following two equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -\left(\mathbf{u} \cdot \nabla\right)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f} \tag{4.2.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{4.2.2}$$

The first term on the right-hand side of equation 4.2.1 is the self-advection of the velocity field, causing velocity to be carried along itself. The pressure term $-\frac{1}{\rho}\nabla p$ causes velocity to move along the pressure gradient, from areas of high pressure to areas of low pressure. The pressure field $p$ is used as a correcting term, ensuring that equation 4.2.2 holds by projecting the velocity field onto its divergence free component. The fluid density $\rho$ can thus be omitted, as the pressure term is only used to ensure mass conservation of the velocity field according to equation 4.2.2.

The last term on the right-hand side of equation 4.2.1 is the external force acting on the velocity field. The external force consists of several separate forces, as shown in the following equation:

$$\mathbf{f} = f_{vorticity} + f_{gravity} + f_{buoyancy}, \tag{4.2.3}$$

where $f_{vorticity}$ is the vorticity confinement force, $f_{gravity}$ is the gravity force due to fuel and exhaust gases, and $f_{buoyancy}$ is the buoyancy force due to heat. These forces are described later in equations 4.2.10, 4.2.12, and 4.2.13.

Like [MK02] and [NFJ02] we have chosen to omit the viscosity term from the differential equation 4.2.1 governing the velocity field, as this term would have negligible effects on the end result. Viscosity, controlling the friction or flow resistance of the fluid, is essential when dealing with thick fluids like syrup or glue, but can be omitted when simulating air and gases like we do.

To solve the Navier-Stokes equations governing the velocity field, we use the stable fluid solver from [Sta99], which was also used by [MK02], [NFJ02], [IMDN05], [KW05], and [BLLR06]. The stability of the solver allows us to take larger timesteps and therefore achieve faster simulations. We described the stable fluid solver in detail earlier in section 2.3.2.

### 4.2.2 Fire density fields

We use the fire density fields as a common term for the scalar fuel gas, exhaust gas, and temperature fields. To create a realistic and moving flame, we use the stable fluid solver and the velocity field to advect the fire density fields throughout the computational domain.

The evolution of a scalar field $d$ over time is described by the following equation from [Sta99]:

$$\frac{\partial d}{\partial t} = -\mathbf{u} \cdot \nabla d + \kappa_d \nabla^2 d - \alpha_d d + S_d, \tag{4.2.4}$$

where $\mathbf{u}$ is the velocity field used to advect the scalar field, $\kappa_d$ is a diffusion constant, $\alpha_d$ is a dissipation rate, and $S_d$ is a source term. The first term on the right-hand side of equation 4.2.4 governs the advection of the scalar quantity $d$ by the velocity field $\mathbf{u}$, the second term governs the diffusion of the scalar quantity $d$, and the third term governs the dissipation of the scalar quantity $d$. The last term $S_d$ is used to increase or decrease the scalar quantity $d$ by applying an external source.

We use slightly modified versions of equation 4.2.4 to describe the evolution of the three fire density fields: fuel gas $g$, exhaust gas $a$, and temperature $T$. The modified equations governing the evolution of respectively the fuel gas field, exhaust gas field, and temperature field are given by 4.2.5, 4.2.6, and 4.2.7. These three equations correspond to the equations used in [MK02].

$$\frac{\partial g}{\partial t} = -\mathbf{u} \cdot \nabla g + \kappa_g \nabla^2 g - \alpha_g g + S_g + C_g \tag{4.2.5}$$

$$\frac{\partial a}{\partial t} = -\mathbf{u} \cdot \nabla a + \kappa_a \nabla^2 a - \alpha_a a + C_a \tag{4.2.6}$$

$$\frac{\partial T}{\partial t} = -\mathbf{u} \cdot \nabla T + \kappa_T \nabla^2 T - \alpha_T T + C_T \tag{4.2.7}$$

The first term on the right-hand side of each equation is the advection term, causing the fire density fields to be carried along by the velocity field. The second term is the diffusion term, simulating the tendency of the densities to spread out. The third term governs the dissipation of the fire density fields, and is controlled by the dissipation rates $\alpha_g$, $\alpha_a$, and $\alpha_T$. Dissipation is natural for exhaust which will condensate over time, and temperature which will convect and radiate. For the fuel gas this constant can be kept low, as fuel gases usually require a very low temperature to condensate. We also have a source term $S_g$ for fuel gas, which is used to inject fuel gas in order to keep the fire burning. The remaining terms $C_g$, $C_a$, and $C_T$ are related to combustion, and are discussed in detail in section 4.2.5.

### 4.2.3 Vorticity confinement

To achieve real-time results we have to use a rather coarse grid in our simulation. In addition, the stable fluid solver suffers from some numerical dissipation, meaning that much of the low-level turbulence that is important for achieving realistic flames is lost. To counterbalance this, we use the vorticity confinement method, also used by [NFJ02], [IMDN05], and [KW05], to find and amplify the vortices that are formed in the velocity field. A vortex is the center of a rotational movement. The vorticity at a given field point thus measures how much rotational movement is present there. The vorticity $\omega$ of the velocity field $\mathbf{u}$ is calculated using the following equation:

$$\omega = \nabla \times \mathbf{u} \tag{4.2.8}$$

Next, the normalized gradient of $|\omega|$, $\mathbf{N}$, pointing from lower to higher concentrations of vorticity, is calculated using the following equation:

$$\mathbf{N} = \frac{\nabla|\omega|}{|\nabla|\omega||} \tag{4.2.9}$$

Finally, we calculate the vorticity force $f_{vorticity}$ as shown in equation 4.2.10. The parameter $\epsilon$ controls the strength of the vorticity confinement force, and $h$ is the distance between two grid cells. Larger values of $\epsilon$ causes more turbulence to be added to the velocity field.

$$f_{vorticity} = \epsilon h \left(\mathbf{N} \times \omega\right) \tag{4.2.10}$$

In order to control the strength of the vorticity confinement force in different areas of the computational domain we can scale each component of the vorticity force according to the following equation:

$$f'_{vorticity} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} f_{vx} \\ f_{vy} \\ f_{vz} \end{bmatrix}, \tag{4.2.11}$$

where $s_x$, $s_y$, and $s_z$ are component scaling factors and $f_{vx}$, $f_{vy}$, and $f_{vz}$ are the components of $f_{vorticity}$. Using this scaling we can increase the vorticity in the x and z directions outside the flame to approximate a turbulent wind field surrounding the flame, causing the flame to flicker and behave more chaotically. More turbulence also helps to simulate parts of the flame detaching from the flame body before fading.

### 4.2.4 Gravity and buoyancy

Gravity and especially buoyancy are important forces when modeling realistic behavior of fire and are used by among other [MK02], [NFJ02], and [IMDN05].

Fuel gas and exhaust gas are pulled down due to a gravity force acting on the velocity field. The gravity force is given by the following equation:

$$f_{gravity} = f_g\,(g + a)\begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{4.2.12}$$

The constant $f_g$ determines the strength of the gravity force, while $g$ and $a$ are the amount of fuel gas and exhaust gas respectively.

The buoyancy force also acts on the velocity field, causing hot air to rise, and is given by the following equation:

$$f_{buoyancy} = f_b\,(T - T_{ambient})\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{4.2.13}$$

The strength of the buoyancy force is determined by the buoyancy constant $f_b$, the temperature $T$, and the ambient temperature constant $T_{ambient}$, which is the temperature of the air surrounding the fire. To create realistic fire the buoyancy force is crucial, as rising air is one of the main causes of the characteristic and turbulent appearance of the flame.

### 4.2.5 Combustion

An important part of the physically based simulation is to take into account the combustion occurring when fuel gas reacts with oxygen creating exhaust gas and heat, as described earlier in section 2.1.1. To model the combustion we choose an approach similar to the one in [MK02]. They do not explicitly model the reaction zone where combustion occurs, instead modeling combustion wherever there is enough fuel, oxygen, and heat.

The combustion will only occur if the temperature is above a given threshold temperature $T_{threshold}$, which corresponds to the ignition temperature of the fuel gas. In contrast to [MK02] we assume that there will always be enough oxygen to react with the fuel gas. This assumption should be realistic for the kind of free-burning diffusion flames we wish to simulate, because these flames usually occur in an environment with sufficient oxygen. We end up with the following equation for the combustion parameter:

$$C = \begin{cases} rbg & \text{if } T > T_{threshold} \\ 0 & \text{if } T \leq T_{threshold} \end{cases}, \tag{4.2.14}$$

where $r$ is the burning rate parameter describing how fast the fuel gas can be burned, $b$ is the stoichiometric mixture describing the amount of oxygen required to burn one unit of fuel, and $g$ is the amount of fuel gas. Equations 4.2.15, 4.2.16, and 4.2.17 use the combustion parameter $C$, and describe the rate of change of respectively the fuel gas, exhaust gas, and temperature due to combustion.

$$C_g = -\frac{C}{b} \tag{4.2.15}$$

$$C_a = C(1 + \frac{1}{b}) \tag{4.2.16}$$

$$C_T = T_0 C \tag{4.2.17}$$

Because we assume there will always be enough oxygen to react with the fuel gas, the parameter $b$ only controls how much oxygen is involved in the reaction, and will not directly affect how fast the fuel is consumed. We see in equations 4.2.16 and 4.2.17 that the higher the value of $b$ the more exhaust gas and heat is produced by the combustion. Because more oxygen is used when the fuel gas reacts, more byproducts are created. In addition, the parameter $T_0$ is used in equation 4.2.17 to control the amount of heat that is produced by the reaction.

### 4.2.6 Mass expansion

The simulation can be extended to take into account the mass expansion that occurs when fuel gas is converted to exhaust gas due to combustion. Our mass expansion approximates the effect of the blue core and ghost fluid approach from [NFJ02] without explicitly representing the blue core surface using a level set. Like [FOA03] did in their simulation of explosions, we modify the mass conservation equation governing the velocity field, which was given earlier in equation 4.2.2, to allow the divergence of the velocity field to be non-zero. This is shown by the following equation:

$$\nabla \cdot \mathbf{u} = C\phi, \tag{4.2.18}$$

where $C$ is the combustion parameter from equation 4.2.14 and $\phi$ is the mass expansion rate. As can be seen, when there is no combustion the divergence is 0 just like before. When combustion occurs and $\phi$ is positive, $\nabla \cdot \mathbf{u} > 0$, meaning that additional fluid is generated. The mass expansion rate is a parameter controlling the rate at which gas expands during combustion.

## 4.3 Boundary conditions

The equations governing the velocity and fire density fields are solved on a per cell basis, which often requires computations based on a cell's neighboring cells. As the cells at the boundary of the domain do not have neighboring cells on all sides, they can not be solved using the same equations as for the interior cells. Thus, special consideration is taken for the boundary cells by setting them based on boundary conditions which specify the behavior of the velocity and fire density fields at the boundaries. We support three main boundary conditions: global wind, open boundary, and closed boundary.

The global wind boundary condition is influenced by the boundary conditions used by [AL04] to simulate falling snow. The boundary cells of the velocity field are all set to a global wind vector, whereas the boundary cells of the fire density fields are set to zero. The result is that the fire simulation is affected by the global wind vector, causing the fire to move realistically due to wind.

The open boundary is set similar to the global wind boundary, except that the velocity boundary cells are set to zero. This boundary condition can be used to model a fire in an open environment.

The closed boundary condition is used to model a closed simulation environment, but it can also be used to increase turbulence for fires that are not actually surrounded by walls. The closed boundary condition has three subcategories, based on the boundary conditions presented in [FM97]: smooth boundary, concrete boundary, and rough boundary. All the subcategories involve setting the boundary cells of the fire density fields to the same value as the corresponding interior cells, ensuring that the fuel gas, exhaust gas, and temperature fields do not leak at the boundary. When setting the velocities at the boundary, we treat the parallel and perpendicular velocity components separately. Figure 4.4 shows how the boundary velocities are set for the smooth, concrete, and rough boundary conditions. The example in the figure is taken from the top boundary of a 2D simulation grid, but the same approach is used for 3D simulation grids.

The parallel velocity component is the velocity component parallel to the boundary plane, whereas the perpendicular velocity component is the velocity component perpendicular to the boundary plane. The perpendicular component of the boundary velocity is set to the inverse of the perpendicular component of the neighboring interior velocity for all three subcategories, which ensures that there is no flow across the boundary. The only way the subcategories differ is in how the parallel components of

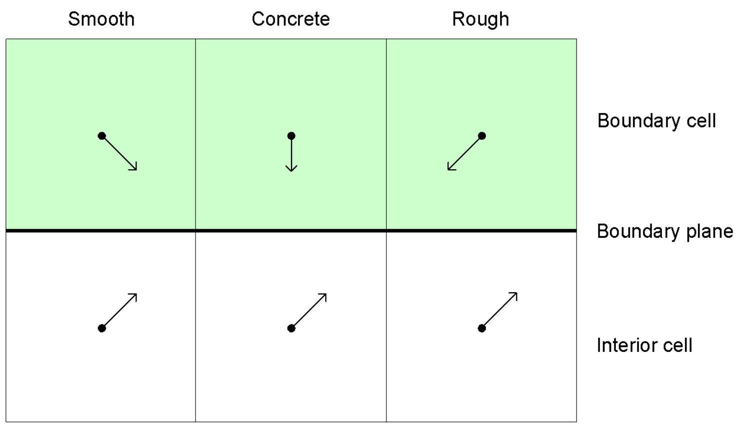

Figure 4.4: Closed boundary conditions for the velocity field.

the boundary velocity are set. The parallel components of the boundary velocities are set to achieve low, medium, and high turbulence near the boundary for respectively smooth, concrete, and rough boundary conditions. The following equation shows how the parallel boundary velocity components are set:

$$\bar{\mathbf{u}}_b = w\bar{\mathbf{u}}_i, \tag{4.3.1}$$

where $\bar{\mathbf{u}}_b$ denotes the parallel component of the velocity at the boundary cell, $\bar{\mathbf{u}}_i$ is the parallel component of the velocity at the corresponding interior cell, and $w$ is a weight that is set individually for the three closed boundary subcategories. For the smooth boundary condition, $w$ is set to 1, ensuring free parallel flow at the boundary. Next, for the concrete boundary condition, $w$ is set to 0, causing a bit of friction at the boundary, which slightly increases turbulence. Finally, for the rough boundary condition, $w$ is set to $-1$, causing even more friction and a higher amount of boundary turbulence.

## 4.4 Sampling from simulation fields

As mentioned earlier in section 4.1.1, field values are defined in the center of cells or voxels of the discretized simulation domain. Thus, when sampling from an arbitrary position in the computational domain, the samples at the discrete cell or voxel positions must be combined to accurately represent in-between values. We use bilinear and trilinear interpolation when sampling from respectively 2D and 3D domains. In addition, we use volumetric extrusion to implicitly sample from a 3D domain by sampling from a set of 2D domains.

### 4.4.1 2D and 3D sampling

Sampling from a 2D domain represented by a grid structure is done by bilinear interpolation. The four nearest neighbors are weighted based on the sample location to produce a weighted average. Similarly, when sampling from a 3D domain trilinear interpolation is used, based on the eight surrounding neighbors.

Using interpolation when sampling enables smooth transitions between the discrete grid points where field values are defined. The smooth transitions are especially beneficial when later visualizing the simulation results.

### 4.4.2 Volumetric extrusion

Performing the simulation on a full 3D voxel grid is quite computationally expensive. Therefore, we have adopted the approach from [RNGF03], who perform several uncoupled 2D simulations and combine them using volumetric extrusion. Figure 4.5 shows an example where two 2D slices are used to define a 3D volume by combining the slices rotationally around the vertical axis. Thus, a complete 3D volume can be represented by a set of 2D slices.

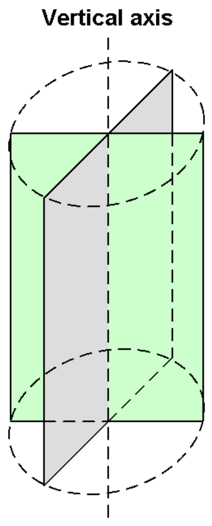

Figure 4.5: Two 2D slices defining a 3D volume.

The advantages of using volumetric extrusion for simulating fire is that the available processing power can be spent more efficiently, achieving a higher quality simulation. As certain fire effects like torches and bonfires are quite rotationally symmetric phenomena, they are good candidates for volumetric extrusion. Only a few slices are simulated, thus each slice can have a larger resolution than the width and height of a 3D simulation.

To sample from the 3D volume implicitly defined by a set of 2D slices, cylindrical interpolation is used between slices and bilinear interpolation is used within slices. Equation 4.4.1 and figure 4.6 show how to compute the sample $S(\mathbf{P})$ at position $\mathbf{P}$. The figure gives a cross section of the slices as seen along the vertical axis, which goes through the origin of the 3D volume. In the equation, $\alpha$ is the angular location of the sample between the two nearest slices, *numslices* is the number of slices, and $S(\mathbf{A})$ and $S(\mathbf{B})$ are the bilinearly interpolated samples from the corresponding positions in the next and previous slice respectively. The corresponding positions have the height $P_y$ and the horizontal distance $|\mathbf{P}_{xz}|$ from the vertical axis. As can be seen from the equation, $\frac{\alpha \times numslices}{\pi}$ is the interpolating factor, ranging from 0 to 1. Thus, when the angle is large $A$ will be the dominating sample, and vice versa.

$$S(\mathbf{P}) = \frac{\alpha \times numslices}{\pi} S(\mathbf{A}) + (1 - \frac{\alpha \times numslices}{\pi}) S(\mathbf{B}) \qquad (4.4.1)$$

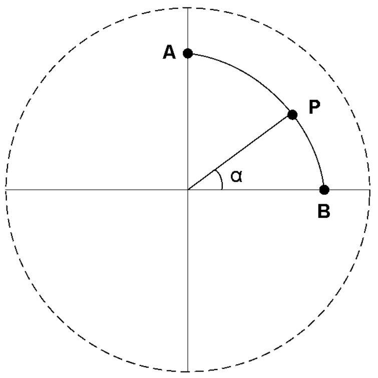

Figure 4.6: Cylindrical interpolation in the xz plane (top view).

Sampling a scalar density value from the implicitly defined 3D volume is straightforward using equation 4.4.1. However, special consideration must be taken when reconstructing 3D velocities from individual 2D velocity slices. The 3D y component can be interpolated directly from the two y components of the nearest 2D velocity slices, whereas both the x and z components of the 3D velocity must be constructed based on two 2D velocity x components. First, the two values $S_x(\mathbf{A})$ and $S_x(\mathbf{B})$ are sampled using bilinear interpolation like before. Next, the signs of the values are adjusted such that they give the relative x velocity with respect to the center of the 2D velocity field, which is where the different 2D slices intersect. Thus, if the x velocity component points towards the vertical axis it gets a negative sign, and if it points away from the vertical axis it gets a positive sign. We end up with the sign-adjusted velocity components $S_{\bar{x}}(\mathbf{A})$ and $S_{\bar{x}}(\mathbf{B})$, which are cylindrically interpolated to get the radial velocity component $V_{radial}$ using equation 4.4.1. Finally, the x and z components of the 3D velocity are computed as shown in equation 4.4.2, by multiplying the interpolated radial velocity $V_{radial}$ with the normalized vector pointing from the origin to the projection of the sample position $\mathbf{P}$ in the xz plane.

$$\mathbf{V}_{xz} = V_{radial} \frac{\mathbf{P}_{xz}}{|\mathbf{P}_{xz}|} \tag{4.4.2}$$

## 4.5 Complete simulation algorithm

The fire simulation evolves the velocity and fire density fields based on the combustion process and a stable fluid solver. The velocity and fire density fields are discretized throughout the computational domain, and each step of the fire simulation is performed for all cells in the discretized grid or voxel structure. Although the equations governing the simulation fields are defined for 3D simulation, corresponding equations can be used in the 2D case. The steps of the complete fire simulation algorithm are shown below:

1. *Velocity force calculation* - Compute and add forces acting on the velocity field.
   (a) *Vorticity confinement force* using the velocity field and equation 4.2.10.
   (b) *Gravity force* using the fuel gas field, exhaust gas field, and equation 4.2.12.
   (c) *Buoyancy force* using the temperature field and equation 4.2.13.
2. *Velocity fluid solver step* - Self-advect and project the velocity field.
   (a) *Advection* using the velocity field and the stable fluid solver to solve the advection part of equation 4.2.1.
   (b) *Projection* using the velocity field and the stable fluid solver to ensure that equation 4.2.2 holds. Optionally, if mass expansion is used, equation 4.2.18 replaces equation 4.2.2.
3. *Fire density force calculation* - Compute and add forces acting on the fuel gas, exhaust gas, and temperature fields.
   (a) *Dissipation* using the fire density fields and dissipation part of equations 4.2.5, 4.2.6, and 4.2.7.
   (b) *Source terms* using a pre-computed fuel gas source field and source part of equation 4.2.5.
   (c) *Combustion forces* using the fire density fields and equations 4.2.15, 4.2.16, and 4.2.17.
4. *Fire density fluid solver step* - Advect and diffuse the fuel gas, exhaust gas, and temperature fields.
   (a) *Advection* using the velocity field and the stable fluid solver to solve the advection part of equations 4.2.5, 4.2.6, and 4.2.7.
   (b) *Diffusion* using the fire density fields and the stable fluid solver to solve the diffusion part of equations 4.2.5, 4.2.6, and 4.2.7.

## 4.6 Summary

We have presented an algorithm for simulating fire. The algorithm models the burning of fuel gas and simulates the evolution of a velocity field specifying the motion of three fire density fields containing fuel gas, exhaust gas, and temperature respectively. The fields are governed by a set of differential equations, which can be solved using Stam's stable fluid solver. A set of forces make the fire behave in a realistic fashion:

- A vorticity force is used to increase the turbulent behavior of the fire.
- Gravity pulls fuel gas and exhaust gas down.
- Buoyancy makes air rise due to heat.
- Combustion turns fuel gas into exhaust gas and heat if the temperature exceeds the fuel's ignition temperature.
- Dissipation ensures that the fire density fields eventually disperse.
- Fuel gas sources are added to the fuel gas field to keep the fire burning.

In addition to these forces, mass expansion manipulates the divergence of the velocity field, and can be used to approximate the expansion that occurs when fuel gas is converted to exhaust gas during combustion. Boundary conditions can be used to achieve various behaviors at the boundary of the simulation domain, including wind effects. The fire simulation can be performed in a 2D or 3D computational domain. Volumetric extrusion can be used to achieve a 2D slice simulation, where a set of 2D simulations together implicitly define a 3D simulation volume.

# Chapter 5

# Visualization of fire

In this chapter we present two different methods for visualizing the fire based on the results from the fire simulation presented in the previous chapter. The first method uses a particle system to visualize the fire, as done by [WLMK02] and [KW05], and in addition we use a second particle system to visualize the emerging smoke. The second method uses a volume renderer to visualize the fire. Simple volume rendering approaches were used by [KCR99], [MK02], and [IMDN05] to visualize fire, but we have chosen to use a recent GPU-based approach which uses ray marching, as described in [KW03]. Finally, we dynamically illuminate the scene using point lights placed in the fire, inspired by the approaches in [IMDN05] and [BLLR06].

The rest of this chapter is organized as follows. First, we give a general description of the coupling between the fire simulation and the methods used to visualize the simulation results, and then we describe the computation of two new fields used in the visualization. Next, we describe our particle system and volume rendering approaches in more detail, and finally we present our dynamic illumination approach.

## 5.1 Overview

In figure 5.1, we show the connections between the fire simulation presented in the last chapter and the visualization approaches presented in this chapter: particle systems, volume rendering, and dynamic illumination.

A pre-computed black-body radiation table is used in combination with the exhaust gas and temperature fields from the fire simulation to compute the fire color field, which represents the fire with its characteristic yellow-orange color. The smoke emerging from the fire is represented by the smoke field, which is also computed based on the exhaust gas and temperature fields from the fire simulation. The fire color and smoke fields combined give a visual representation of the fire simulation results.

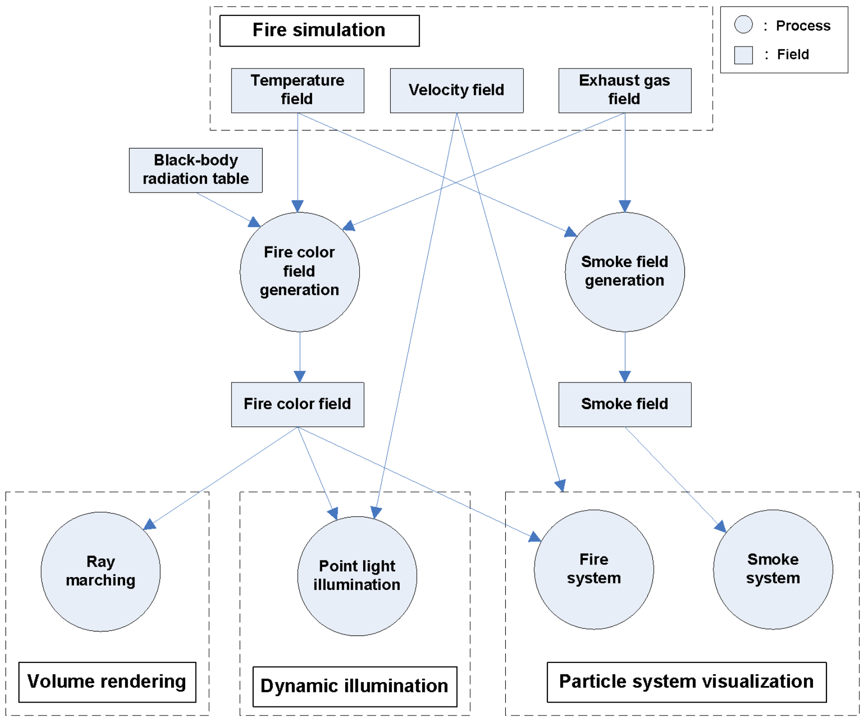

Figure 5.1: Visualization processes and fields.

We visualize the fire using two particle systems: one for the fire using the fire color field and one for the smoke using the smoke field. These two fields are used to sample colors for the particles in the two particle systems, whereas the velocity field from the fire simulation is used to move the particles.

The volume renderer however uses only the fire color field to visualize the fire. A ray marching approach is used to trace rays through the 3D volume represented by the fire color field, and color values are sampled at discrete steps along the rays.

The point light positions are updated in a similar fashion as the particles in the particle systems using the velocity field from the fire simulation. The brightnesses of the point lights are sampled from the fire color field.

## 5.2 Fire color field

The fire color field is the result of combining the exhaust gas and temperature fields from the fire simulation to represent the actual fire with realistic colors, and is thus defined in the same 2D or 3D computational domain as the rest of the fire simulation fields. The yellowish-orange color we associate with fire comes from the black-body radiation emitted by the hot exhaust gas [NFJ02], and using a black-body radiation table for computing the fire color field is therefore essential to create realistic fire colors. Black-body radiation in combination with visualization of fire is also used by [WLMK02] and [IMDN05].

### 5.2.1 Black-body radiation

We use Planck's formula for black-body radiation given by equation 5.2.1 in order to calculate the intensity radiated by the hot exhaust gas. Planck's formula was presented earlier in section 2.1.2.

$$B_\lambda(T) = \frac{2\pi hc^2}{\lambda^5 \left(e^{\frac{hc}{\lambda kT}} - 1\right)} \tag{5.2.1}$$

By using the wavelengths of red, green, and blue light and the temperature of the exhaust gas we calculate the three intensities $B_{red}$, $B_{green}$, and $B_{blue}$. These intensities have a very high dynamic range, whereas the resulting color should have a limited range suitable for display on traditional computer monitors. To map the intensities onto a limited range, we use the exponential mapping function from [Mat97], as shown in the following equation:

$$I = 1 - e^{\frac{-B}{B_{average}}}, \tag{5.2.2}$$

where $B$ is the original intensity, and $B_{average}$ is a constant controlling the overall brightness. The resulting intensity $I$ will be in the range $[0, 1\rangle$.

Equations 5.2.1 and 5.2.2 are too computationally expensive to be calculated each frame. Instead, we use them to pre-compute black-body radiation color values for a user specified range of temperatures. The pre-computed color values for various temperatures are stored in a one-dimensional lookup table.

### 5.2.2 Computing the fire color field

The fire color field is computed using the exhaust gas and temperature fields in combination with the pre-computed black-body radiation lookup table. The same computation is performed for each cell in the discretized computational domain in which the fire color field is defined. Equation 5.2.3 shows how the color **c** in the fire color field is computed, based on the temperature $T$, the amount of exhaust gas $a$, and a temperature scaling factor $T_{scale}$, which is used to control the resulting brightness of the fire. *lookup* is the black-body radiation lookup table.

$$\mathbf{c} = a \times \mathit{lookup}\left(T_{scale}T\right) \tag{5.2.3}$$

The color of the hot exhaust is retrieved using the lookup table, and the resulting fire color is scaled based on the amount of exhaust gas in the cell being computed. A higher exhaust gas density causes brighter colors. The left part of figure 5.2 shows an example 2D fire color field of dimensions 64x96 computed using equation 5.2.3.

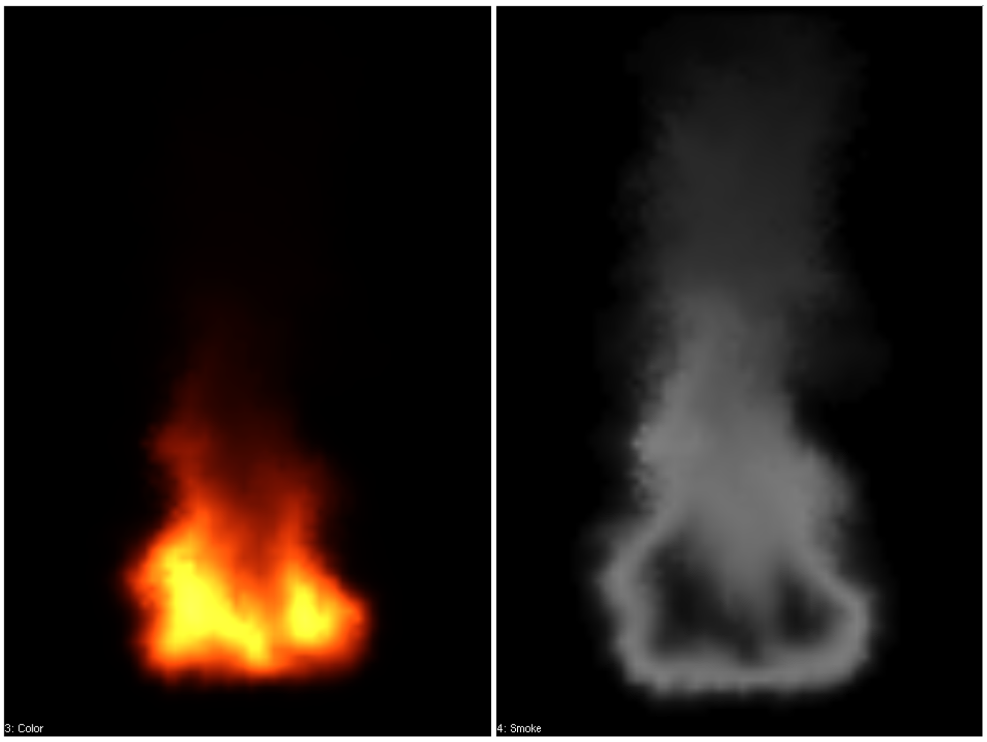

Figure 5.2: 2D fire color field (left) and smoke field (right) from the same timestep.

## 5.3 Smoke field

What we perceive as smoke is actually exhaust gas which has cooled down. The smoke field represents the smoke generated in the fire and is like the fire color field computed using the exhaust gas and temperature fields from the fire simulation. Like the fire color field, it is defined in the same 2D or 3D computational domain as the fire simulation fields. By computing a separate smoke field instead of trying to incorporate smoke into the fire color field, we get more control over the appearance and amount of smoke produced in the fire.

In order to model the amount of smoke generated based on the temperature, we use a fall-off curve as presented in figure 5.3. Based on the temperature this curve gives a smoke generation factor in the interval $[0, 1]$, and lower temperatures lead to higher smoke generation factors. Using the fall-off curve we arrive at the following equation to calculate the amount of smoke in a cell in the discretized computational domain in which the smoke field is defined:

$$\mathbf{s} = a \times s_{density} \times curve\left(T_{scale}T\right), \tag{5.3.1}$$

where $a$ is the amount of exhaust gas, $s_{density}$ is used to control the density of the smoke, $T$ is the temperature, and $T_{scale}$ is a constant used to scale the temperature to fit into the input range of the fall-off curve. From the fall-off curve we see that little or no smoke is generated when the temperature is high. However, when the temperature sinks, an increasing amount of smoke is generated, approximating the appearance of smoke as the exhaust gas starts to cool down. The smoke is therefore present around the base of the fire, and should surround the core of the fire color field. The right part of figure 5.2 shows an example 2D smoke field of dimensions 64x96 computed using equation 5.3.1. The fire color field and smoke field in the figure are from the same simulation timestep. The amount of smoke corresponds to the brightness of the greyscale color. The color of the smoke is user-specified and set in addition to $\mathbf{s}$ for each cell in the discretized computational domain.

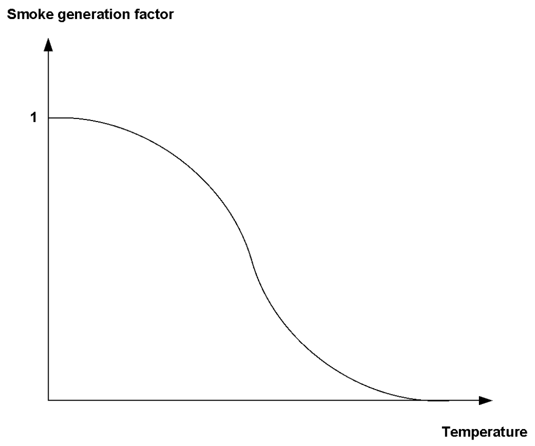

Figure 5.3: Smoke fall-off curve.

## 5.4 Particle systems

One of our methods for visualizing the fire uses particle systems defined in the same computational domain as the fire simulation, similar to the approaches in [WLMK02] and [KW05]. We use separate particle systems for visualizing fire and smoke. Each particle represents a small fire or smoke element respectively and has a set of associated variables: spawn position, current position, initial spawn delay, current velocity, and color. Spawn position and initial spawn delay are given at the beginning of the simulation, whereas the other variables are dynamically updated. A particle's color is specified by an RGBA color value. The particles are spawned at the base of the fire and set to follow the fire simulation's velocity field, and the fire and smoke particle colors are sampled from the fire color and smoke fields respectively. Finally, particles are respawned when no longer visible.

### 5.4.1 Particle simulation

Initially the particles are not visible. Each particle has an individual spawn delay, after which its position is set to its spawn position. This spawn delay is used to avoid the periodic effect that could otherwise occur before the simulation has time to settle, due to most of the particles spawning and respawning at approximately the same time.

At the beginning of each particle simulation step, particle velocities are sampled from the fire simulation velocity field based on the current particle positions as shown in the following equation:

$$\mathbf{v}_i = \mathit{sample}(\mathit{velocity}, \mathbf{x}_i), \tag{5.4.1}$$

where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle $i$ respectively and *velocity* is the velocity field from the fire simulation. The sampling of velocities was described earlier in section 4.4. A simple Euler step is later used to update a particle's position as described by the following equation:

$$\mathbf{x}'_i = \mathbf{x}_i + \mathbf{v}_i \Delta t, \tag{5.4.2}$$

where $\Delta t$ is the timestep.

Like particle velocities, particle colors $\mathbf{c}_i$ are sampled from the fire color or smoke fields based on the updated particle positions, as shown in the following equation:

$$\mathbf{c}_i = \mathit{sample}(\mathit{field}, \mathbf{x}_i) \tag{5.4.3}$$

When a particle's intensity drops below a specified threshold, it is respawned by resetting its position to its spawn position. A minimum initial lifetime ensures that the particle is not respawned before it has had a chance to enter the fire.

### 5.4.2 Particle rendering

Particles are rendered using textured view-aligned quads centered at the particle positions. Like in [WLMK02] and [Hol03], texturing is used to create more low-level detail, allowing larger particles and reducing the number of particles needed. In order to create the illusion of light-absorbing and scattering smoke, and semi-transparent and emitting fire, different blending modes are used to combine the colors of overlapping quads. We avoid the high cost of sorting by using appropriate blending modes and particle textures.

Equation 5.4.4 shows the blending equation used for fire colors, which is independent of the order in which the fragments are blended [1]. $C_{dst}$ is the color of the destination, i.e. the previous fragment in the framebuffer, and $C_{src}$ is the color of the current fragment of the particle quad being rendered. The first part of the right-hand side of the equation is the emission of the fire, and the second part is responsible for making the fire semi-transparent, by reducing the contribution of the background color on the result.

$$C_{dst} = C_{src} + C_{dst} * (1 - C_{src}) \tag{5.4.4}$$

Similarly, equation 5.4.5 shows the blending equation used for smoke, where $A_{src}$ is the alpha value of the current particle fragment being rendered, corresponding to $\mathbf{s}$ computed in equation 5.3.1, and $C_{src}$ is the user specified smoke color modulated by the smoke texture splat. We use the same alpha blending approach as [Hol03] used for smoke visualization with a particle system. In this case we model absorption as well as scattering.

$$C_{dst} = A_{src} * C_{src} + (1 - A_{src}) * C_{dst} \tag{5.4.5}$$

The smoke particles are rendered before the fire particles, to make sure the smoke particles do not obscure the fire. Fire has a much higher intensity than smoke, and thus should be rendered last.

The left and right parts of figure 5.4 show examples of textures used for fire and smoke particles respectively. It is important that the textures fade out towards the sides, in order to create fluent transitions when the particles are blended. The textures used are greyscale and specify the brightness of the fire or smoke colors at different parts of the particle quad. The actual color is fetched from the fire color field or smoke field respectively.

### 5.4.3 Visualization algorithm using particle systems

After the velocity and fire density fields have been calculated in a step of the fire simulation algorithm, as shown in section 4.5, the fields in turn are used by two particle systems to visualize the fire and the emerging smoke.

[1]Equation 5.4.4 can be written as: $D_n = \sum_{i=1}^{n} S_i - \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} S_i S_j + \sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n} S_i S_j S_k + \cdots + (-1)^{n-1}\sum$ products of permutations of length n-1 $+ (-1)^{n}\sum$ products of permutations of length n, where $D_n$ is the color value in the framebuffer after $n$ fragments have been blended and $S_i$ is the color value of fragment $i$. We assume that $D_0 = 0$. This equation shows that equation 5.4.4 is independent of the order the fragments are blended.

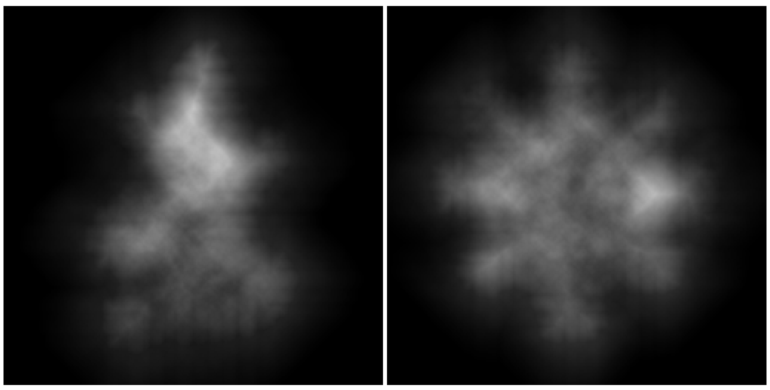

Figure 5.4: Fire splat (left) and smoke splat (right) used for particle rendering.

First, the smoke particle system is updated and visualized according to the following algorithm:

1. Compute the smoke field from the temperature and exhaust gas fields using equation 5.3.1.
2. For each particle, update its velocity by sampling from the fire velocity field based on the current particle position using equation 5.4.1.
3. For each particle, update its position based on the sampled velocity from step 2 using equation 5.4.2. Respawn the particle if below the visibility threshold.
4. For each particle, update its color by sampling from the smoke field based on the new particle position using equation 5.4.3.
5. For each particle, render it as a view-aligned quad using a texture splat colored by the sampled color from step 4. Blend particles using equation 5.4.5.

Next, the fire particle system is updated and visualized according to a corresponding algorithm:

1. Compute the fire color field from a pre-computed black-body radiation table and the temperature and exhaust gas fields using equation 5.2.3.
2. For each particle, update its velocity by sampling from the fire velocity field based on the current particle position using equation 5.4.1.
3. For each particle, update its position based on the sampled velocity from step 2 using equation 5.4.2. Respawn the particle if below the visibility threshold.
4. For each particle, update its color by sampling from the fire color field based on the new particle position using equation 5.4.3.

5. For each particle, render it as a view-aligned quad using a texture splat colored by the sampled color from step 4. Blend particles using equation 5.4.4.

## 5.5 Volume rendering

We also have the option of using a volume rendering approach for visualizing the fire. Our volume rendering approach is based on ray marching in the volume defined by the bounding box surrounding the computational domain of the fire simulation. Ray marching as done by [KW03] was presented briefly in section 3.6.1. The volume renderer is used to visualize the fire color field computed as described in section 5.2.2.

### 5.5.1 Ray calculation

The rays used to march through the volume are computed by tracing rays originating at the camera position and moving through each of the pixels of the screen's viewport. For each pixel, the rays are intersected against the bounding box of the computational domain. A set of sample view rays are shown in figure 5.5. First, the intersection point at which the view ray enters the bounding box is determined. This intersection point correspond to the origin $r_{origin}$ of the ray later used to trace the fire color volume. The intersection point at which the view ray exits the bounding box is then computed. Using the trace ray origin $r_{origin}$ and the exiting intersection, the direction $r_{direction}$ and length $r_{length}$ of the trace ray are computed. The origin $r_{origin}$, direction $r_{direction}$, and length $r_{length}$ of the trace ray are stored in order to be used during the ray marching.

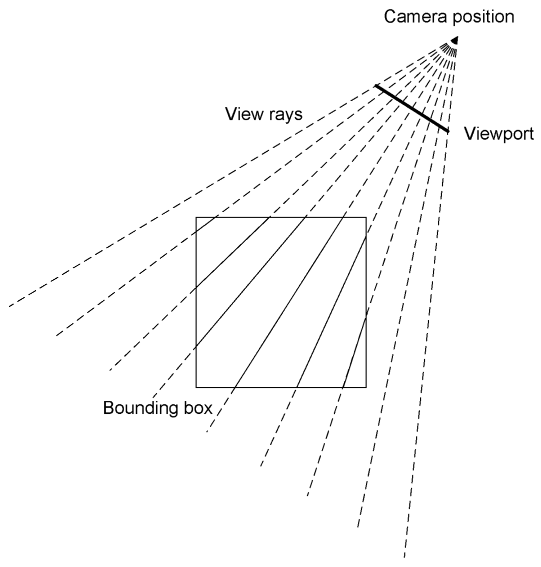

Figure 5.5: Sample view rays used in the ray calculation (top view).

### 5.5.2 Ray marching

For each pixel whose view ray intersects the bounding box, the ray origin $r_{origin}$ and direction $r_{direction}$ are now used to step through the fire color volume. A variable $t$, as shown in equation 5.5.1, is initially set to 0 and then repeatedly incremented as shown in equation 5.5.2 to sample at discrete steps throughout the volume. $r_{current}$ denotes the current position in the volume and *stepsize* is the distance between subsequent sample positions in the volume.

$$r_{current} = r_{origin} + r_{direction} \times t \tag{5.5.1}$$

$$t = t + stepsize \tag{5.5.2}$$

While stepping through the volume, a color value is accumulated by sampling from the fire color field as explained earlier in section 4.4. Equation 5.5.3 shows how the accumulated color $C_{acc}$ is blended with the color $C_{sample}$ sampled at $r_{current}$. This is the same blending function used for the fire particle system. The first part of the right hand side of the equation is the emission and the second part is the absorption in the hot exhaust gas.

$$C_{acc} = C_{sample} + C_{acc}(1 - C_{sample}) \tag{5.5.3}$$

The marching continues until $r_{current}$ is outside of the bounding box, which is when $t > r_{length}$. Finally, the accumulated color is blended with the color $C_{dst}$ from the framebuffer as shown in the following equation:

$$C_{dst} = C_{acc} + C_{dst}(1 - C_{acc}) \tag{5.5.4}$$

### 5.5.3 Visualization algorithm using volume rendering

After a step of the fire simulation algorithm, shown earlier in section 4.5, the exhaust gas and temperature fields are used to compute the fire color field. The fire color field is in turn used by the volume renderer to visualize the fire. The volume rendering algorithm is as follows:

1. Compute the fire color field from a pre-computed black-body radiation table and the temperature and exhaust gas fields using equation 5.2.3.
2. For each pixel in the viewport, calculate the origin $r_{origin}$, direction $r_{direction}$, and length $r_{length}$ of a trace ray by finding the intersections of the view ray with the bounding box of the computational domain.

3. For each pixel in the viewport, use the trace ray calculated in step 2 to march through the fire color volume, accumulating a color using the blending function in equation 5.5.3.

4. For each pixel in the viewport, blend the resulting color with the background using equation 5.5.4.

## 5.6 Dynamic illumination

A fire visualization is not complete without accounting for the illumination of the surrounding scene caused by the fire. Dynamic illumination has been performed by among other [NFJ02], [IMDN05], and [BLLR06], as discussed earlier in section 3.2.4.

Our approach for achieving dynamic illumination involves approximating the illumination from the fire by a set of stationary or moving point lights. The moving point lights are spawned at the base of the fire and follow the velocity field just like the particles in the particle systems. The point light intensities are computed based on base light colors modulated by the brightness of colors sampled from the fire color field. To illuminate the scene based on the point lights, we use the simple Phong illumination model.

### 5.6.1 Basic light model

The emitting radiant energy from the fire is modeled by a set of point light sources positioned inside the fire. Each point light has a position $P_l$ and an emitted light intensity as shown in the following equation:

$$I_l^* = \begin{bmatrix} I_{rl}^* \\ I_{gl}^* \\ I_{bl}^* \end{bmatrix}, \tag{5.6.1}$$

where $I_l^*$ is the intensity of a point light $l$ and $I_{rl}^*$, $I_{gl}^*$, and $I_{bl}^*$ are the RGB components of $I_l^*$. The illumination $I_l$ at a point $P$ after attenuation is then given by:

$$I_l = \frac{1}{|P_l - P|^2} I_l^* \tag{5.6.2}$$

The point lights are used to illuminate the objects in the scene based on the Phong illumination model, as described in [HB04]. Phong is a local illumination model suitable for real-time visualization. A point on a surface to be illuminated is affected by global ambient, diffuse, and specular lighting. The global ambient lighting is the same for all objects, and is a simple approximation of the global diffuse interreflections between the different illuminated surfaces in the scene. The effect of the global ambient lighting on a point $P$ on a surface is given by the following equation:

$$I_{amb} = k_a I_a, \tag{5.6.3}$$

where $k_a$ is the ambient-reflection coefficient of the material of the surface and $I_a$ is the intensity of the global ambient light.

The diffuse lighting on the other hand accounts for the light reflected off the surface equally in all directions. The diffuse lighting component at a point $P$ on a surface is computed as shown in the following equation:

$$I_{diff} = \begin{cases} k_d I_l(N \cdot L), & \text{if } N \cdot L > 0 \\ 0, & \text{if } N \cdot L \leq 0 \end{cases}, \tag{5.6.4}$$

where $k_d$ is the diffuse-reflection coefficient of the material of the surface, $I_l$ is the intensity of the point light at point $P$, $N$ is the normalized surface normal, and $L$ is the normalized vector toward the point light source. Figure 5.6 gives a visual explanation of these terms.

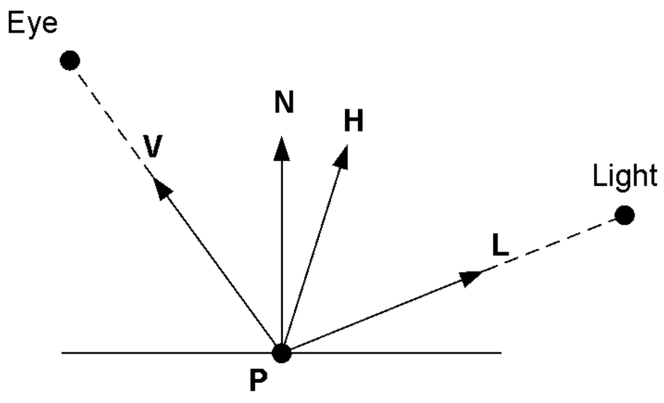

Figure 5.6: Illumination of a point $P$.

The last part of the Phong illumination model is the specular lighting, which approximates light that is scattered from a surface in a concentrated region around the specular reflection direction. This direction equals the reflection of $L$ through $N$. The specular lighting component at a point $P$ on a surface is given by the following equation:

$$I_{spec} = \begin{cases} k_s I_l(N \cdot H)^{n_s}, & \text{if } N \cdot H > 0 \text{ and } N \cdot L > 0 \\ 0, & \text{if } N \cdot H \leq 0 \text{ or } N \cdot L \leq 0 \end{cases}, \tag{5.6.5}$$

where $k_s$ is the specular-reflection coefficient of the material of the surface, $H$ is the normalized vector that is halfway between $L$ and $V$, the normalized vector toward the eye position. Finally, $n_s$ is the specular-reflection exponent determined by the type of surface to be illuminated.

The total illumination effect on the point $P$ using one point light is then given by:

$$I = I_{amb} + I_{diff} + I_{spec} \tag{5.6.6}$$

For multiple point lights we use the following equation instead:

$$I = I_{amb} + \sum_{l=1}^{n} (I_{l,diff} + I_{l,spec}), \tag{5.6.7}$$

where $n$ is the number of point lights and $I_{l,diff}$ and $I_{l,spec}$ are the diffuse and specular contributions from light $l$ respectively.

### 5.6.2 Moving point lights

When a flame is simulated, its shape changes over time and so does the photometric distribution of the flame [BLLR06]. Because it is very difficult to capture photometric solids from different shapes of turbulent flames, it is inconvenient to adopt the method used by [BLLR06]. They visualize simple and almost stationary flames of candles and oil lamps and can therefore use a single photometric solid to model the emission of the flames. Approximating the photometric distribution of a turbulent fire with a single photometric can not be done in the same way because of the large variations of the fire. Instead, the positions of the point lights that we use to model the emitting radiant energy of the fire can be updated according to the motion of the fire, ensuring that the point light positions always reflect the variations of the fire. By using the velocity field from the fire simulation, the positions of the point lights can be updated in a similar fashion as the particles in the particle system using the following equation:

$$P_l' = P_l + sample(velocity, P_l)\Delta t, \tag{5.6.8}$$

where $P_l$ is the position of light $l$, *velocity* is the velocity field, and $\Delta t$ is the timestep. When a light moves outside the computational domain of the fire simulation it is respawned at the base of the fire in the same way as particles.

### 5.6.3 Point light intensities

As we do not use a photometric solid to model the emission of the point lights, we instead base the intensities of the lights on the intensity of the simulated fire. [IMDN05] calculate the light intensities using a temperature distribution and Planck's formula for black-body radiation. By sampling the light brightnesses from the fire color field we get a similar result. The difference is that [IMDN05] have a point light positioned in each voxel of the discretized computational domain leading to high rendering times, while we allow a user specified number of lights to model the emitting radiant energy of the fire. Based on the position of each point light, we sample a color from the fire color field using the following equation:

$$C_l = sample\left(fireColor, P_l\right), \tag{5.6.9}$$

where $C_l$ is the sampled color, $fireColor$ is the fire color field, and $P_l$ is the position of light $l$. To better control the desired color emitted by the fire we specify a base color $C_l^b$ for each light and use the brightness of the sampled color $C_l$ to scale the base color. The final intensity of light $l$ is then computed using the following equation:

$$I_l^* = C_l^b brightness\left(C_l\right) \tag{5.6.10}$$

### 5.6.4 Illumination algorithm

After the velocity and fire density fields have been updated in a step of the fire simulation algorithm, as shown in section 4.5, the fields in turn are used to update the set of point lights. If moving point lights are used, their positions are updated by sampling from the velocity field. In a similar fashion, the intensity of the point lights are updated by sampling from the fire color field. The point lights are finally used to illuminate other objects in the scene.

The following algorithm is used to update the point lights and illuminate the scene:

1. For each moving point light, update its position by sampling from the fire velocity field based on the current light position and move it an Euler step forward using equation 5.6.8.
2. For each point light, update its intensity by sampling from the fire color field based on the current light position using equation 5.6.10.
3. Illuminate the scene objects based on the point lights and the Phong illumination model.

## 5.7 Summary

We have presented particle systems and volume rendering as two methods for visualizing the results from the fire simulation. As a base for both methods we calculate a fire color field using black-body radiation based on the exhaust gas and temperature fields. In addition, for particle systems we calculate a smoke field also based on the exhaust gas and temperature fields. We use two particle systems whose particle colors are sampled from the fire color field and smoke field respectively, based on the particle positions. In addition, the particle positions are moved by the velocity field from the fire simulation. The particles are rendered as semi-transparent textured quads using different blending modes for smoke and fire particles. The smoke particles are rendered before the fire particles, such that the fire is not obscured by the smoke. When volume rendering we only visualize the fire color field. The visualization is done by marching rays through the fire simulation volume, sampling intensities from the fire color field and blending them together. Finally, the resulting color is blended with the framebuffer.

In addition to the main fire visualization, we have presented an approach for dynamically illuminating the scene surrounding the fire. The illumination from the fire is approximated by a set of point lights. These point lights can either be stationary or set to follow the velocity field from the fire simulation. The brightness of each point light is sampled from the fire color field, and we use the brightness to scale the point light's base color. Finally, we render the objects of the scene using the Phong illumination model to light the objects based on the point light sources.

# Chapter 6

# Implementation

In this chapter we describe how the theory presented in chapter 4 and chapter 5 is implemented by utilizing the processing power of the GPU. We first give a general overview of the implementation, presenting the main modules and their interconnections. Next, we describe the framework we have implemented for performing general-purpose computations on the GPU. We continue by presenting the fluid solver implementation and then the implementation of the fire simulation. Next, we present the implementation of the visualization approaches: particle systems, volume rendering, and dynamic illumination. We then present the complete implementation and how the simulation and visualization parts are combined. Finally, we present two extra features which extend the fire simulation to model realistic fire movement and dynamic wind.

## 6.1 Overview

Figure 6.1 gives an overview of the modules that are part of the implementation. The main module is *GPU Computational Framework*, which is used by all the other modules and contains the basic functionality for performing general-purpose computation on the GPU. *Fluid Solver* is a general-purpose module for performing fluid simulations and is used by *Fire Simulation*. Together, the *Fluid Solver* and *Fire Simulation* modules comprise the implementation of the fire simulation algorithm presented in section 4.5. The two visualization algorithms presented in section 5.4.3 and 5.5.3 are realized by the *Particle System* and *Volume Renderer* modules respectively. These two modules are loosely coupled to the *Fire Simulation* module in that they use the results from the fire simulation as base for visualization. The dynamic illumination algorithm presented in section 5.6.4 is realized by the *Dynamic Illumination* module. This module uses the *Particle System* module to update the positions of the moving point lights and the fire color field from the *Fire Simulation* module to update the point light intensities. The rest of this chapter explains each of the modules in figure 6.1 in more detail.

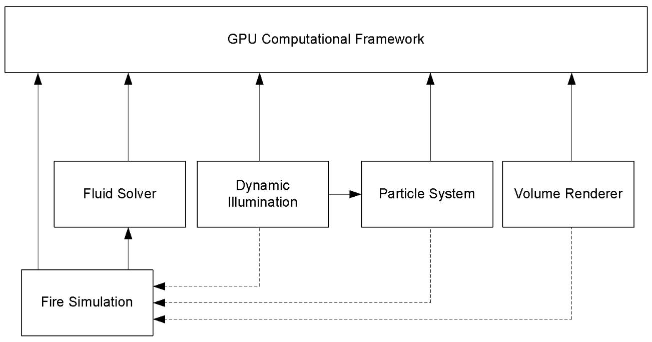

Figure 6.1: Implementation modules.

## 6.2 GPU computational framework

To be able to perform fluid simulations and other general-purpose computations on the GPU, we have implemented a framework for general-purpose computation, consisting of a set of classes representing general-purpose computation concepts. General-purpose computation on the GPU was introduced earlier in section 2.4. Our framework is an interface for accessing important concepts like fragment and vertex programs, texture computations, and flat 3D textures. All fragment and vertex programs used in the implementation of the fire simulation and visualization algorithms are written in Cg (C for graphics) [Cg06], a high-level language for programming the GPU.

In the rest of this section we first present the main classes of our framework for general-purpose computation on the GPU. Next, we give an example of a typical use of the framework to give a more intuitive understanding of the computational flow and how the classes are used. Finally, we describe the implementation details of flat 3D textures.

### 6.2.1 Class overview

Figure 6.2 shows the main classes in our GPU computational framework. All of these classes are somehow connected to the task of performing general-purpose computation on the GPU, although not all are limited to this. A selection of the most important member functions of each class are presented in the class diagram. We give a brief presentation of each class, focusing on functionality and usage.

#### 6.2.1.1 CgContext

`CgContext` is responsible for initializing a valid Cg context, and determines which fragment and vertex profiles are supported by the graphics hardware. Finally, it lets the user enable and disable the fragment and vertex profiles respectively.

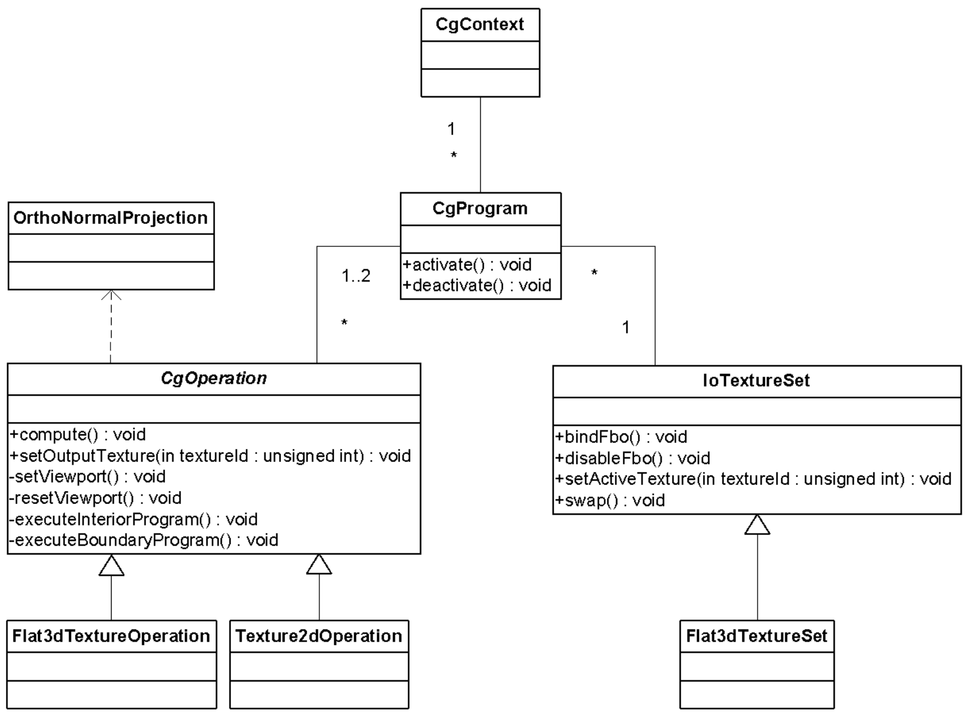

Figure 6.2: Class diagram: GPU computational framework.

#### 6.2.1.2 IoTextureSet

`IoTextureSet` represents a set of textures which can be used as input or output by texture operations performed in `CgOperation` subclasses. Each texture gets an associated texture identifier which is used to refer to it. Textures are used as output by enabling a framebuffer object and attaching the texture to it. Enabling and disabling the framebuffer object is done by the two static member functions `bindFbo()` and `disableFbo()` respectively, whereas textures are attached to the framebuffer object using the member function `setActiveTexture()`. Additionally, `IoTextureSet` offers member functions for accessing the texture data of the contained textures.

The textures contained in an `IoTextureSet` instance are divided into primary and secondary textures. Primary textures all have the same dimensions and format, whereas the secondary textures can be of different dimensions and formats. Swapping (or ping-ponging) is supported by adding an additional temporary texture of the same dimensions and format as the primary textures. To swap the currently active texture with the temporary texture the member function `swap()` is used. Swapping is needed when the same texture is used both for input and output by a texture operation. During the texture operation, the temporary texture is then used as output instead of the active texture, and after the texture operation has completed the two textures are swapped. Secondary textures are not supported for swapping, but can be imported from other

`IoTextureSet` instances in order to share a texture between several instances.

#### 6.2.1.3 Flat3dTextureSet

`Flat3dTextureSet` is a specialization of `IoTextureSet` for flat 3D textures. It determines an efficient layout for texture tiles and creates domain, slice, and offset lookup textures needed by Cg programs operating on flat 3D textures. In addition, it provides functionality for writing voxel grid values to flat 3D textures without needing to know the internal representation. Flat 3D textures are explained in more detail later in section 6.2.3.

#### 6.2.1.4 CgProgram

`CgProgram` is a simple wrapper around a Cg fragment or vertex program. An instance of `CgProgram` loads and initializes a single Cg program, and makes it easy to activate and deactivate it by using the member functions `activate()` and `deactivate()` respectively. In addition, `CgProgram` contains support for setting various uniform parameters like floats, vectors, and textures, which are used by the fragment or vertex program when performing computations. A uniform parameter does not vary from fragment to fragment or vertex to vertex, but is constant throughout a rendering pass. If the Cg program needs to read from textures, a reference to an `IoTextureSet` instance must be supplied when initializing the `CgProgram` instance. The actual OpenGL texture identifiers are fetched from the `IoTextureSet` instance using the `IoTextureSet` texture identifiers.

#### 6.2.1.5 CgOperation

`CgOperation` is an abstraction of fragment programs which perform texture operation. It uses one `CgProgram` instance for interior operations and optionally one `CgProgram` instance for boundary operations. The interior operations and boundary operations are performed for texels representing the interior cells and boundary cells respectively. The distinction between interior cells and boundary cells was shown earlier in figures 4.1 and 4.2 for 2D and 3D domains respectively. The results from the operations are written to an output texture using render-to-texture functionality on the GPU. The user sets which output texture to use through the member function `setOutputTexture()`, and the `CgOperation` instance determines whether swapping is required (if the output texture is also an input texture for either of the interior or boundary `CgProgram` instances). To perform the operation represented by an instance of `CgOperation` the member function `compute()` is offered, but it can only be called through subclasses as `CgOperation` is an abstract class. A call to `compute()` corresponds to a rendering or computation pass. `CgOperation` is also responsible for setting the correct viewports needed when operating on the textures.

#### 6.2.1.6 Texture2dOperation

`Texture2dOperation` is a specialization of `CgOperation` for standard 2D texture operations which operate on a single rectangular texture. A quad is drawn to process each of the interior quad fragments using the fragment program represented by the interior `CgProgram` instance. If boundary computation is activated, a one pixel wide frame surrounding the quad is drawn to process the boundary fragments using the fragment program represented by the boundary `CgProgram` instance.

#### 6.2.1.7 Flat3dTextureOperation

`Flat3dTextureOperation` is a specialization of `CgOperation` for performing operations on flat 3D textures. A quad is drawn for each tile in the flat 3D texture to process all the interior quad fragments using the fragment program represented by the interior `CgProgram` instance. If boundary computation is activated, one pixel wide frames surrounding each quad are drawn to process the boundary fragments using the fragment program represented by the boundary `CgProgram` instance. In addition, for 3D volume domains the front and back quad fragments are also processed using the boundary `CgProgram` instance.

#### 6.2.1.8 OrthoNormalProjection

`OrthoNormalProjection` is a convenient class for use in combination with `CgOperation`. Upon instantiation it sets the current projection matrix to orthonormal and the current modelview matrix to identity. When the instance leaves scope the old projection and modelview matrices are reset. This reduces the burden on the user, as `CgOperation` and subclasses assume orthonormal projection and identity modelview matrices when drawing quads or frames.

### 6.2.2 Computational flow

Figure 6.3 shows a sequence diagram for a general-purpose computation on a 2D texture using a fragment program running on the GPU. As mentioned earlier, the fragment program is executed for each texel in the texture. We assume that the goal of the computation is to add the content of two input textures, writing the result to an output texture.

`Adder` represents the class responsible for the computation, which is invoked by the call to `run()`. First, a new instance of `OrthoNormalProjection` is created, making sure the OpenGL projection matrix is set to orthonormal and modelview matrix to identity. Then, the framebuffer object is activated by the call `bindFbo()` ensuring that the results from the fragment program are written to the active output texture

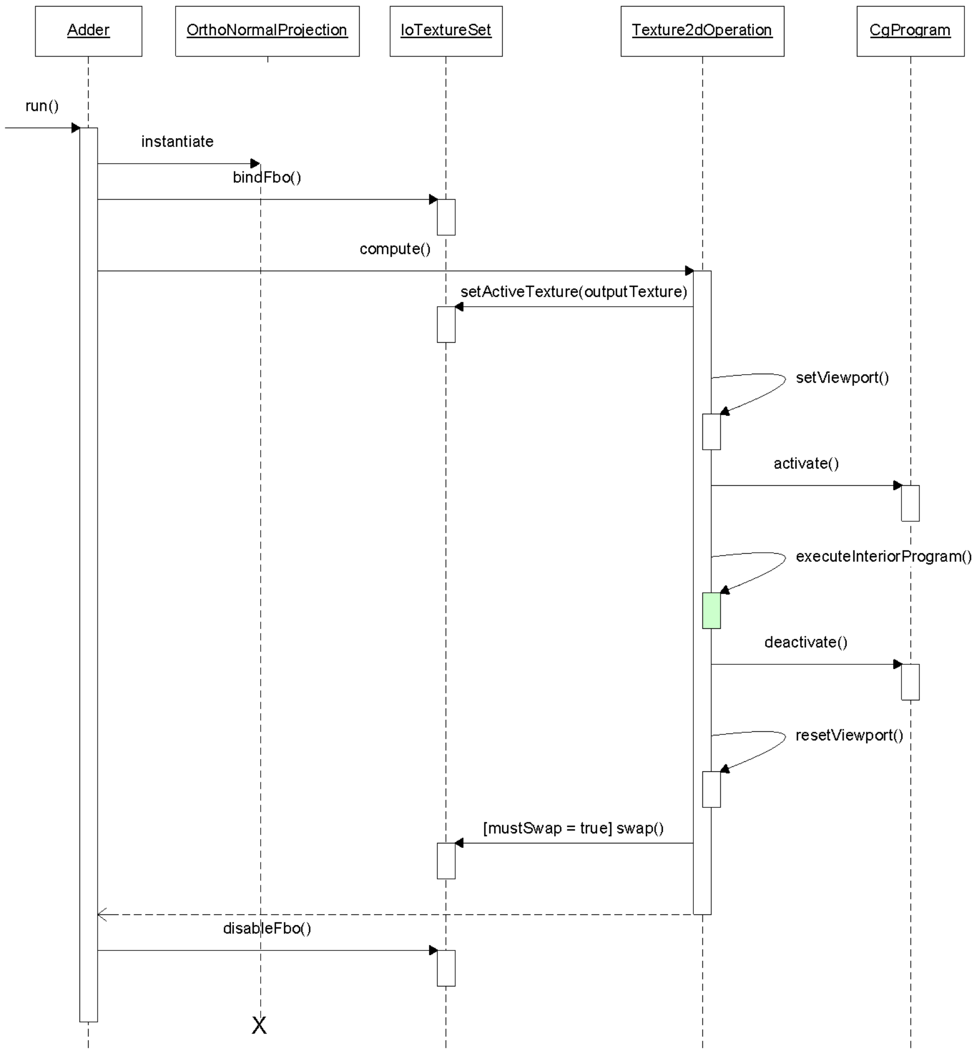

Figure 6.3: Sequence diagram: GPU computational framework.

and not the framebuffer. The computation itself is initiated by the call `compute()` to `Texture2dOperation`. The output texture specified in the `Texture2dOperation` instance is set as the active output texture using the framebuffer object. Next, the OpenGL viewport is set according to the width and height of the output texture. This makes sure that width times height fragments are generated in the rasterizer when the quad is drawn. Before the quad is drawn to initialize the computation the `CgProgram` instance is activated. This activation includes binding the fragment program to the GPU, enabling the fragment profile, and setting and enabling all textures used as input textures by the fragment program. When the quad is drawn, shown as a green box in figure 6.3, the fragment program is executed for all fragments generated as explained in section 2.4.3 and shown in figure 2.7. For each fragment the corresponding two texel values are fetched from the two input textures using the interpolated texture coordinates, and the result is written to the corresponding texel of the active output texture. After the quad is drawn, the `CgProgram` instance is deactivated and the viewport is reset to its previous state. If the output texture equals one of the input textures used by the fragment program, swapping is performed in the `IoTextureSet` instance. Finally, the framebuffer object is disabled and the OpenGL projection and modelview matrices are reset to previous states in the destructor of `OrthoNormalProjection`.

### 6.2.3 Flat 3D textures

Flat 3D textures were introduced by [HBSL03] in the context of cloud simulations on the GPU. They are needed as GPUs do not currently support rendering directly to 3D textures. A flat 3D texture is the representation of a 3D voxel grid in a 2D texture. The 3D voxel grid is split into slices along the z direction, and these slices are stored as tiles in the flat 3D texture. An example flat 3D texture is shown in figure 6.4. Note that the number of texels in the flat 3D texture must be at least as high as the number of voxels in the 3D voxel grid, as each voxel must be represented by a corresponding texel.

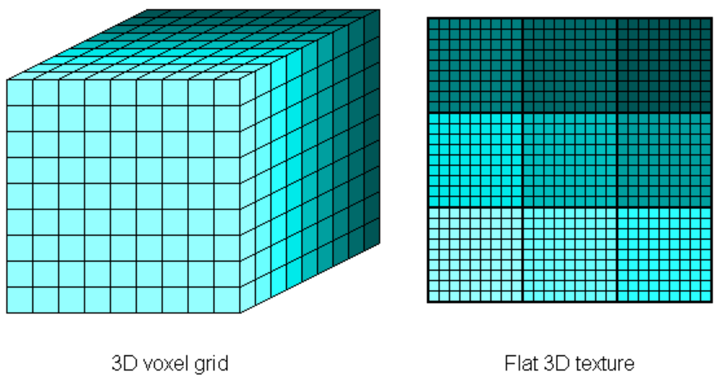

Figure 6.4: Flat 3D texture layout.

When performing 2D slice simulation with volumetric extrusion we also use flat 3D textures, as this makes it easier to sample from the 3D volume implicitly defined by the 2D slices. In this case the tiles of the flat 3D texture represent the individual 2D slices and not slices of a 3D voxel grid, and the texels represent cells instead of voxels.

When performing computations on flat 3D textures representing a 3D voxel grid, it is often necessary to access voxels from neighboring slices. To achieve this we use a pair of lookup textures representing domain coordinates and tile offsets. We refer to these textures as the domain lookup texture and offset lookup texture respectively. The domain lookup texture is a 2D texture of the same dimensions as the flat 3D texture, mapping from local coordinates to global coordinates, and the offset lookup texture is a 1D texture mapping from z coordinates to 2D tile coordinates. Local coordinates are the 2D texture space coordinates used to access the flat 3D textures, whereas the global coordinates are 3D voxel space coordinates. The 2D tile coordinates give the offset of the lower left corner of the 2D tile in the flat 3D texture.

For flat 3D textures representing individual 2D slices of a 3D volume implicitly defined through volumetric extrusion, a slice lookup texture is needed to perform the sampling discussed in section 4.4.2. Each slice is divided into two half-slices along the vertical axis, as shown in figure 6.5. This division is done because which half-slice to sample from depends on the sample location. The slice lookup texture contains tile coordinates and half-slice directions for each half-slice in a counter-clockwise direction around the vertical axis of the implicit 3D volume. In this case the tile coordinates give the offset of the bottom center of the 2D tile corresponding to the slice that the given half-slice belongs to. The half-slice direction is respectively $-1$ and 1 for each of the half-slices in a slice. Table 6.1 gives an overview over all the lookup textures and their inputs and outputs.

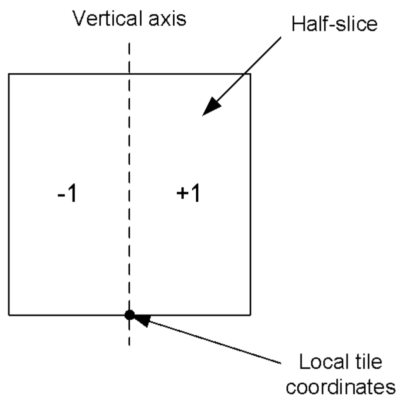

Figure 6.5: A slice divided into two half-slices.

| **Lookup texture** | **Input** | **Output** |
|---|---|---|
| Domain lookup | Local 2D coordinates | Global 3D coordinates |
| Offset lookup | Z coordinate | Local 2D tile coordinates |
| Slice lookup | Half-slice index | Local 2D tile coordinates and half-slice direction |

Table 6.1: Lookup textures used for flat 3D textures.

## 6.3 Fluid solver

The fluid solver module consists of a set of classes each responsible for a part of the fluid solver. The classes `Advection`, `Projection`, `Diffusion`, and `VorticityConfinement` correspond to the fluid solver processes shown earlier in figure 4.3. In addition, we use the `ForceAdder` class to add forces to velocity and density fields. When initializing these classes, a boundary fragment program must be supplied to specify the behavior at the boundary. The fluid solver is used to solve the Navier-Stokes equations 4.2.1 and 4.2.2 governing the velocity field and the advection and diffusion terms of equations 4.2.5, 4.2.6, and 4.2.7 affecting the fuel gas, exhaust gas, and temperature fields respectively.

The fluid solver is based on the stable fluid solver, which we described in 2.3.2, and all computations are performed in fragment programs on the GPU. The GPU implementation is partially based on [Har04], who describe the GPU implementation of the stable fluid solver in 2D. We have extended the implementation to 3D and 2D slice domains using flat 3D textures. The fragment programs used for 3D and 2D slice simulations are encapsulated by `CgProgram` and `Flat3dTextureOperation` instances from the GPU computational framework, which is utilized to perform the fluid solver operations. The fragment programs can be seen in appendix D.1 and appendix D.2 for respectively 3D and 2D slice simulations.

### 6.3.1 Textures

The parts of the fluid solver perform various operations on velocity and density fields. These fields have to be discretized and are represented as textures on the GPU in order to perform computations on them. In table 6.2 we show the textures used by the fluid solver and what is stored in the different texture channels. The divergence, pressure, and vorticity textures are used internally by the fluid solver to perform computations, and we describe how the contents of these textures are calculated in the following sections. The other textures used by the fluid solver are supplied by the user. The quantity texture shown in the table is a common denominator for the velocity and density textures. Likewise, the force texture can contain both velocity and density forces.

| **Texture** | **Description** | **Red** | **Green** | **Blue** | **Alpha** |
|---|---|---|---|---|---|
| Velocity | Contains the vector components of a discretized velocity field. | X-direction | Y-direction | Z-direction (3D only) | |
| Divergence | Contains the divergence values of a discretized velocity field. | Amount of divergence | | | |
| Pressure | Contains the values of a discretized pressure field. | Amount of pressure | | | |
| Vorticity | Contains the vorticity vector components of a discretized velocity field. | Vorticity in the X-direction | Vorticity in the Y-direction | Vorticity in the Z-direction | |
| Density | Contains the values of up to four discretized density fields. | Amount of density *a* | Amount of density *b* | Amount of density *c* | Amount of density *d* |
| Quantity | Contains the vector components of a discretized vector field or the values of up to four discretized density fields. | X-direction or amount of density *a* | Y-direction or amount of density *b* | Z-direction or amount of density *c* | Amount of density *d* |
| Force | Contains the vector components of a discretized vector force field or the force values of up to four discretized density force fields. | X-direction or density *a* force | Y-direction or density *b* force | Z-direction or density *c* force | Density *d* force |

Table 6.2: Overview of fluid solver textures and the content of the four texture channels.

### 6.3.2 Advection

`Advection` takes two textures: a velocity texture representing a velocity field and either a velocity or density texture representing the quantity to advect. We solve the advection part of equation 4.2.1 using the implicit semi-Lagrangian approach presented in [Sta99], which basically views the discretized fields as particles distributed uniformly in space. Each particle is traced using the velocity field to find the position the particle would end up at if advected backwards in time. Finally, the values of the texels closest to this position are interpolated to find the new value for the current particle. This approach can be used both for advecting up to four density fields and for self-advecting a velocity field.

In the 2D case advection is rather simple. For a given cell the velocity $v$ is read from the velocity texture. The velocity is multiplied by the timestep $\Delta t$ and subtracted from the current position $x$, resulting in the position $x'$ which the current value would end up at if advected back in time over one timestep. $x'$ is calculated as shown in the following equation:

$$x' = x - v\Delta t \tag{6.3.1}$$

As the position of the cells correspond directly to the texture coordinates of the texture, the new value is found by bilinearly interpolating from the quantity texture at the computed position $x'$.

Advection on flat 3D textures is a bit more complicated. Unlike for 2D domains, $x'$ is computed using the domain lookup texture and must be clamped to the 3D computational domain. The reason no clamping is required for the 2D case is that texture fetches outside the boundary of a texture can be automatically clamped to the texture's boundary on the GPU. Using flat 3D textures this must be done explicitly, as reading outside the boundary of the xy plane could result in fetching a value from a neighboring z slice. After clamping, trilinear interpolation is used in the quantity flat 3D texture. For the implementation of 3D advection see `advect3d()`.

Explicit clamping of texture coordinates is also required for 2D slice simulation, where several 2D slices are stored in a single flat 3D texture. In this case however, only bilinear interpolation is required. The `advectSlice()` fragment program is used for advection in the 2D slice case.

### 6.3.3 Projection

The class `Projection` uses the GPU to ensure the non-divergence condition of an incompressible velocity field, as shown earlier in equation 4.2.2. As in [Sta99], the projection process is based on the Heimholtz-Hodge Decomposition stating that every vector field is the sum of a non-divergent vector field and the gradient of a scalar field.

In the context of the stable fluid solver, this scalar field is the pressure of the velocity field. Projection involves finding this pressure field and then subtracting its gradient from the velocity field.

Computing the pressure field corresponding to a divergent velocity field requires solving the Poisson equation shown earlier in equation 2.3.7. The Poisson equation is discretized into a system of linear equations and then solved on the GPU using a Jacobi iteration. The Jacobi iteration step for a single pressure grid cell $x_i$ is shown in equation 6.3.2, where $b$ is the divergence of the corresponding velocity field cell. When solving the Poisson equation $\alpha$ is set to 1 and $\beta$ is set to $\frac{1}{4}$ or $\frac{1}{6}$ for 2D slice or 3D simulation respectively.

$$x_i^n = \frac{b + \alpha \sum_{j \in neighbors(x_i)} x_j^{n-1}}{\beta} \tag{6.3.2}$$

The divergence of the velocity field is computed in `divergence2d()` or `divergence3d()` for 2D slice or 3D simulation respectively, and the divergence is stored in the temporary divergence texture. Next, a number of Jacobi iterations are required to compute the pressure texture. Each Jacobi iteration step is performed in `jacobi2d()` or `jacobi3d()`. Typically, 20 iterations are sufficient for realistic behavior. After the pressure texture has been computed, its gradient is subtracted from the velocity field in `subtractGradient2d()` or `subtractGradient3d()` respectively.

To support mass expansion, which allows the velocity field to be divergent as shown earlier in equation 4.2.18, we have added a divergence force texture, allowing the user to modify the temporary divergence texture before the pressure texture is computed. The `setDivergenceForceTexture()` member function of `Projection` is used to set the texture containing the divergence force.

### 6.3.4 Diffusion

The diffusion part of the fire density equations in section 4.2.2 is handled by the `Diffusion` class, which takes a density texture representing up to four density fields, the diffusion constants of the fields, and the voxel size of the discretized domain. Because fields are represented as textures and packing is used to store different quantities in the color channels of the textures, different diffusion constants can be specified for the different channels. Additionally, a timestep is supplied each time the diffusion computation is run.

Diffusion is solved using implicit backward integration, as shown in equation 2.3.5, resulting in a system of linear equations which like projection is solved using Jacobi iteration. The Jacobi equation has the same structure as equation 6.3.2, but now $x_i$ and $x_j$ are cells from the resulting density fields, $b$ refers to a cell from the old density field, and $\alpha$ and $\beta$ are set as shown in the following equations:

$$\alpha = \frac{\kappa \Delta t}{v^N} \tag{6.3.3}$$

$$\beta = 1 + |neighbors|\alpha, \tag{6.3.4}$$

where $\kappa$ is the diffusion constant, $\Delta t$ is the timestep, $v$ is the voxel size, $N$ is the dimension (2 or 3) and $|neighbors|$ is 4 or 6 for 2D slice or 3D simulations respectively.

As for projection, the Jacobi iteration is performed in the `jacobi2d()` or `jacobi3d()` fragment programs and 20 iterations are typically enough.

### 6.3.5 ForceAdder

The `ForceAdder` class takes two textures: a force texture and a quantity texture representing a vector field or up to four density fields. The force texture is added to the quantity texture using `addForce3d()` for the 3D case and `addForce2d()` for the 2D slice case. For each fragment processed, the force $f$ is fetched from the force texture, multiplied with a timestep, and added to the current value $x$ fetched from the quantity texture. The following equation is used:

$$x' = x + f\Delta t \tag{6.3.5}$$

### 6.3.6 VorticityConfinement

`VorticityConfinement` is used to add turbulence to a velocity field by means of vorticity confinement, which was explained in section 4.2.3. The class takes a velocity texture and a velocity force texture as input, and the resulting vorticity confinement force is written to the given force texture.

When `VorticityConfinement` is used for 3D fluid simulation the two fragment programs `generateVorticity3d()` and `addVorticity3d()` are used. `generateVorticity-3d()` is used to compute the vorticity $\omega$ using equation 4.2.8, and the results are written to a temporary vorticity texture. By using the definitions of the gradient and the cross-product operator we arrive at the following equation to compute the vector components of the vorticity $\omega$:

$$\omega = \nabla \times \mathbf{u} = \left(\frac{\partial \mathbf{u}_z}{\partial y} - \frac{\partial \mathbf{u}_y}{\partial z}\right)\hat{\mathbf{x}} + \left(\frac{\partial \mathbf{u}_x}{\partial z} - \frac{\partial \mathbf{u}_z}{\partial x}\right)\hat{\mathbf{y}} + \left(\frac{\partial \mathbf{u}_y}{\partial x} - \frac{\partial \mathbf{u}_x}{\partial y}\right)\hat{\mathbf{z}} \tag{6.3.6}$$

To compute the partial derivatives in equation 6.3.6 we use the finite difference form as in [Har04]. The temporary vorticity texture is then used by `addVorticity3d()` to compute the vorticity confinement force $f_{vorticity}$ using equations 4.2.9 and 4.2.10. The gradient of the length of the vorticity vector ($\nabla|\omega|$) is also computed using the finite difference form from [Har04]. Both fragment programs use the domain and offset lookup textures to fetch values from the velocity and temporary vorticity flat 3D textures.

The fragment programs `generateVorticity2d()` and `addVorticity2d()` are used instead when `VorticityConfinement` is used for 2D slice simulation. These programs are a bit simpler than the ones used in the 3D case.

## 6.4 Fire simulation

In figure 6.6 we give a visual presentation of the processes involved in the GPU implementation of a single pass of the fire simulation algorithm presented in section 4.5. The fire simulation builds on the fluid solver to perform the main fire simulation. The fields from the algorithm, as well as the fire color and smoke fields, are discretized and represented as flat 3D textures. The input textures and the output texture where the results are written are given for each GPU computation. In addition, the figure shows the order in which the computations are performed. The computations are separated into three steps: two main simulation steps utilizing the fluid solver and a color texture calculation step. The fire color and smoke texture calculations are logically part of the visualization, but they are implemented as part of the fire simulation for convenience. The processes of figure 6.6 are performed in the member function `doRun()` of the `Fire3dSimulator` class. When `Fire3dSimulator` is instantiated, the dimensions of the discretized computational domain, the type of the simulation (2D slice or 3D), and the boundary conditions used for the velocity field and fire density fields must be supplied. The discretized computational domain dimensions determine the dimensions of the primary textures of the `Flat3dTextureSet` instance used by the `Fire3dSimulator` instance to store the simulation textures.

The fire specific fragment programs used by `Fire3dSimulator` are general enough that they can be used for both 2D slice and 3D simulation. This is mainly because these fragment programs perform only local operations and do not need to look at neighboring cells. Thus, the main difference between performing 2D slice and 3D simulations is that the fluid solver classes are initialized using different fragment programs. All the fragment programs used directly by `Fire3dSimulator` are encapsulated by `CgProgram` and `Flat3dTextureOperation` instances from the GPU computational framework, which is utilized to perform the fire simulation operations. The fire specific fragment programs are shown in appendix D.3.

At the start of the simulation, all the fire simulation textures are empty. Thus, a requirement for getting a burning fire is that the fuel source texture contains fuel sources. In addition, to start the reaction we use an ignite step where the reaction threshold is set below zero for a short time interval. Setting of fuel sources and igniting is handled by member functions `setFuelSources()` and `ignite()` of `Fire3dSimulator`.

In the rest of this section we will first present the main fire simulation textures used

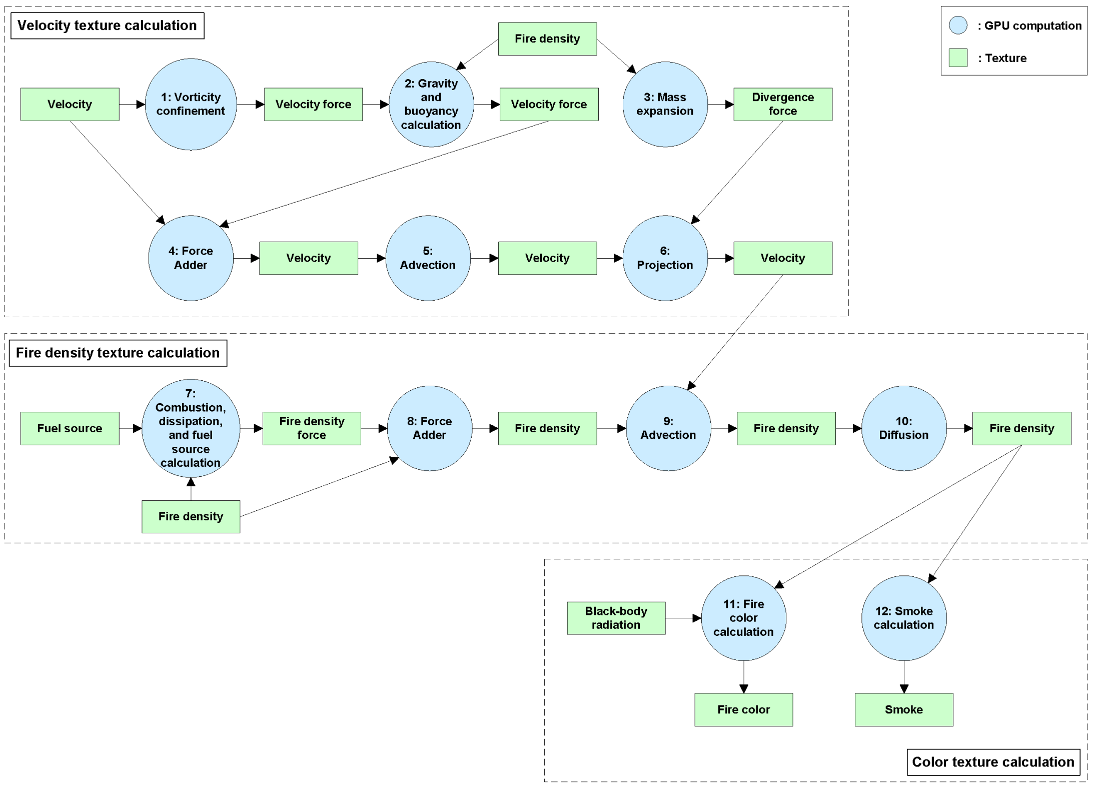

Figure 6.6: The processes involved in the GPU implementation of the fire simulation algorithm.

by `Fire3dSimulator` and then discuss the implementation of each of the three steps in figure 6.6.

### 6.4.1 Simulation textures

In section 4.5 we presented the algorithm for the fire simulation. The algorithm operates on the velocity field and the fire density fields, and in order to implement this algorithm completely on the GPU we discretize the fields and represent them as flat 3D textures. The dimensions of the discretized computational domain, given when `Fire3dSimulator` is instantiated, are used to specify the dimensions of the flat 3D textures used. For 3D simulation the discretized computational domain is split into slices along the z direction, and these slices are stored as tiles in the flat 3D texture. The dimensions of these tiles thus correspond to the width and height of the discretized computational domain, and the number of tiles corresponds to the depth of the discretized computational domain. For 2D slice simulation with volumetric extrusion, the tiles of the flat 3D texture represent the individual 2D slices, and the dimensions of these tiles correspond to the width and height of the discretized computational domain. When using 2D slice simulation, the depth of the implicitly defined 3D domain always corresponds to the width of the discretized computational domain as the 3D domain is rotational symmetric.

To be able to create textures that correspond to the dimensions of the discretized computational domain we use `GL_TEXTURE_RECTANGLE_NV` as texture target, ensuring that the textures can have non-power of two widths and heights. In addition, the texture coordinates associated with the textures range from 0 to *width* and 0 to *height* instead of being normalized, simplifying the implementation. We also need textures with high floating point precision to avoid numerical dissipation and achieve simulations with sufficient accuracy. `GL_FLOAT_RGBA32_NV` is therefore used as the internal format of the textures, ensuring 32-bit precision for each of the four texture channels.

Each texel in the textures we use consists of an RGBA value, meaning that up to four density fields or a single vector field can be represented in a single texture. As we have three fire density fields, these can be represented by one texture by packing the exhaust gas, fuel gas, and temperature fields in the red, green, and blue texture channels respectively. A texel in the texture thus contains the amount of exhaust gas, fuel gas, and temperature at that grid cell. The same approach is used when packing the x, y, and z directions of the velocity field into one texture. A texel then contains the x, y, and z directions of the vector field at that grid cell. Packing several fields into a single texture greatly decreases the number of textures and rendering passes needed.

In table 6.3 we show the textures used in the implementation and what is stored in the different texture channels. All the textures are stored and managed by a `Flat3dTextureSet` instance. The black-body radiation texture, fire color texture, and smoke texture are used to store RGB color values in the interval $[0, 1]$, while the other simulation textures store non-normalized values in the different texture channels.

These values vary in accordance with the simulation parameters used.

In the algorithm implementation we compute the velocity forces and fire density forces explicitly and then store the values in the velocity force texture and fire density force texture respectively. This is needed as several forces are accumulated before finally being added to the velocity texture and fire density texture. We also use a texture for the fuel gas sources, representing the location and amount of fuel gas to be injected per second into the fuel gas part (green channel) of the fire density texture.

### 6.4.2 Velocity texture calculation

The forces acting on the velocity field are vorticity confinement, buoyancy, and gravity. The vorticity confinement force is computed and written to the velocity force texture using an instance of `VorticityConfinement`, which is part of the fluid solver. Next, the fragment program `calculateVelocityForces()` adds buoyancy and gravity to the velocity force texture. The buoyancy and gravity forces are calculated based on the exhaust gas and temperature components of the fire density texture according to equations 4.2.13 and 4.2.12 respectively. The `reactionThreshold` parameter is used by `calculateVelocityForces()` in order to scale the vorticity confinement force in the x and z directions when there is no combustion reaction. Equation 4.2.11 is used to perform this scaling. The left and right parts of figure 6.7 show respectively the x and y components of the velocity force texture for a single timestep of a 2D slice simulation with one slice of dimensions 64x96. The pixels appearing in the figure correspond to the texels of the simulation texture, and thus the cells of the discretized computational domain. Bright intensities correspond to positive velocity forces while dark intensities correspond to negative velocity forces. For the x component figure, positive velocity is defined from left to right and for the y component figure, positive velocity is defined from bottom to top. Finally, the mass expansion is calculated by the fragment program `calculateDivergenceForces()` according to equation 4.2.18, and the results are written to the divergence force texture.

After the velocity force texture has been computed, the fluid solver is used to update the velocity texture. First, the forces are added to the velocity texture using an instance of `ForceAdder`. Following this, the velocity texture is self-advected using an `Advection` instance and projected using a `Projection` instance. The x and y components of the updated velocity texture are shown in respectively the left and right parts of figure 6.8. As for the velocity force texture, bright intensities correspond to positive velocity while dark intensities correspond to negative velocity. It is apparent from the y component texture that there is a lot of upward motion in the velocity field. In the x component texture we can see a lot of turbulence, especially at the edges of the flame.

| **Texture** | **Description** | **Red** | **Green** | **Blue** | **Alpha** |
|---|---|---|---|---|---|
| Velocity | Contains the vector components of the discretized velocity field. | X-direction | Y-direction | Z-direction (3D only) | |
| Velocity force | Contains the vector components of the discretized velocity force field used to accumulate the forces acting on the velocity field (vorticity confinement, buoyancy, and gravity). | X-direction | Y-direction | Z-direction (3D only) | |
| Fire density | Contains the values of the discretized fire density fields. | Amount of exhaust gas | Amount of fuel gas | Temperature | |
| Fuel source | Contains the values of the discretized fuel gas source field. | | Injected fuel gas per second. | | |
| Fire density force | Contains the force values of the discretized fire density force fields used to accumulate the forces acting on the fire density fields (combustion, dissipation, and sources). | Exhaust gas force | Fuel gas force | Temperature force | |
| Divergence force | Contains the force values of the discretized divergence force field used for mass expansion. | Divergence force value | | | |
| Black-body radiation | Contains the colors of the precomputed black-body radiation table. | Red color value | Green color value | Blue color value | |
| Fire color | Contains the colors of the discretized fire color field. | Red color value | Green color value | Blue color value | Fire color brightness |
| Smoke | Contains the colors and smoke amounts of the discretized smoke field. | Red color value | Green color value | Blue color value | Amount of smoke |

Table 6.3: Overview of fire simulation textures and the content of the four texture channels.

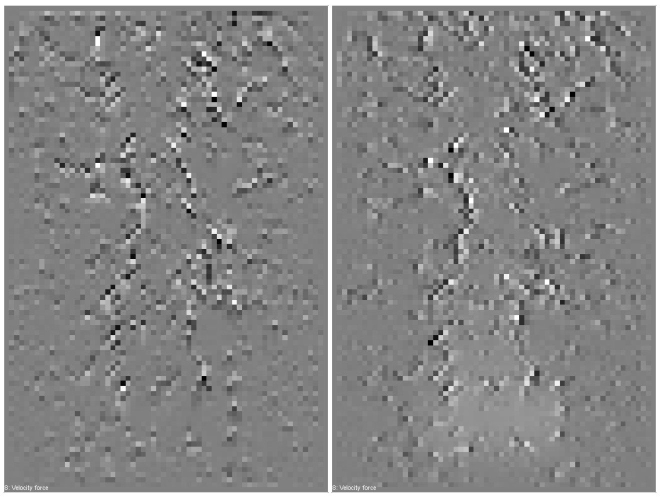

Figure 6.7: The x component (left) and y component (right) of a 2D velocity force texture.

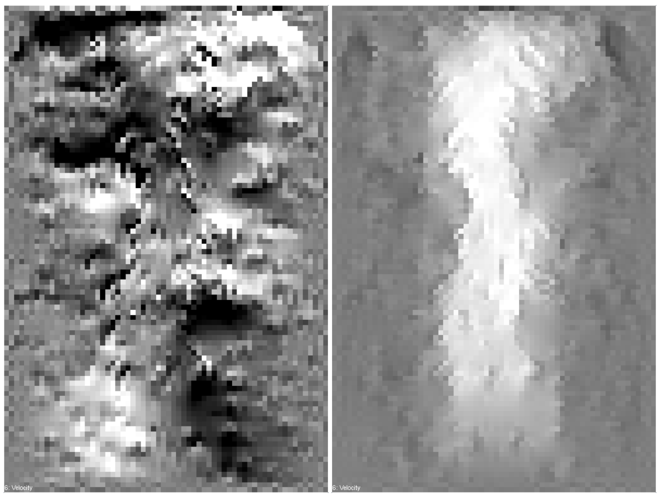

Figure 6.8: The x component (left) and y component (right) of a 2D velocity texture.

### 6.4.3 Fire density texture calculation

The fire density texture is affected by dissipation, fuel sources, and combustion. All of these terms are computed and written to the fire density force texture in the fragment program `calculateDensityForces()`. Dissipation is calculated based on the fire density texture and the dissipation rate parameters. This calculation corresponds to the dissipation part of equations 4.2.5, 4.2.6, and 4.2.7. The fuel gas sources are added from the fuel source texture and correspond to the source part of equation 4.2.5. Finally, combustion forces are computed based on the burning rate, output heat, stoichiometric mixture, and reaction threshold parameters. These forces are calculated using equations 4.2.15, 4.2.16, and 4.2.17. The exhaust gas, fuel gas, and temperature parts of the fire density force texture are shown in respectively the left, center, and right parts of figure 6.9. Bright intensities correspond to additive forces, while dark intensities correspond to subtractive forces. As can be seen, the fuel gas is combusted and the exhaust gas and temperature are dissipated as the fire rises.

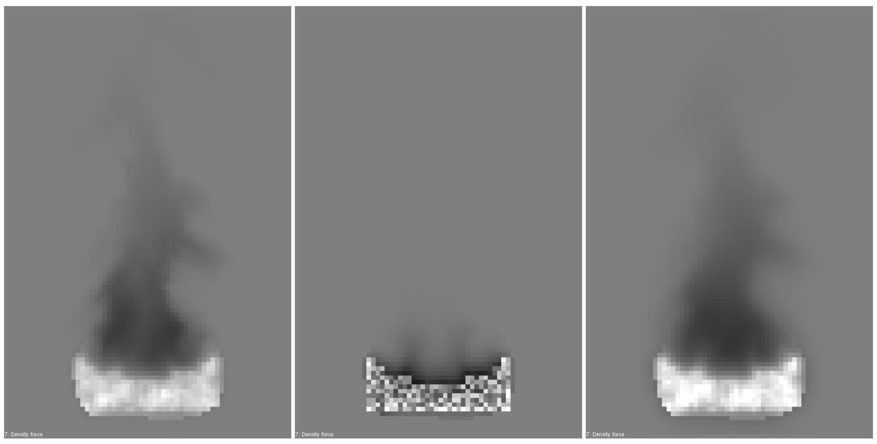

Figure 6.9: The exhaust gas (left), fuel gas (center), and temperature (right) parts of a 2D fire density force texture.

After the fire density force texture has been computed, the fluid solver is used to update the fire density textures. First, the forces are added to the fire density texture using an instance of `ForceAdder`. Following this, the fire density texture is advected using an `Advection` instance and the velocity texture. At last the fire density texture is diffused using an `Diffusion` instance. The left, center, and right parts of figure 6.10 show the exhaust gas, fuel gas, and temperature parts of the updated fire density texture. The pixel intensities correspond to the amount of density or temperature present.

### 6.4.4 Color texture calculation

After a step of the fire simulation, the fire density texture is used to compute the fire color texture and smoke texture to be used in the visualization. It is convenient to

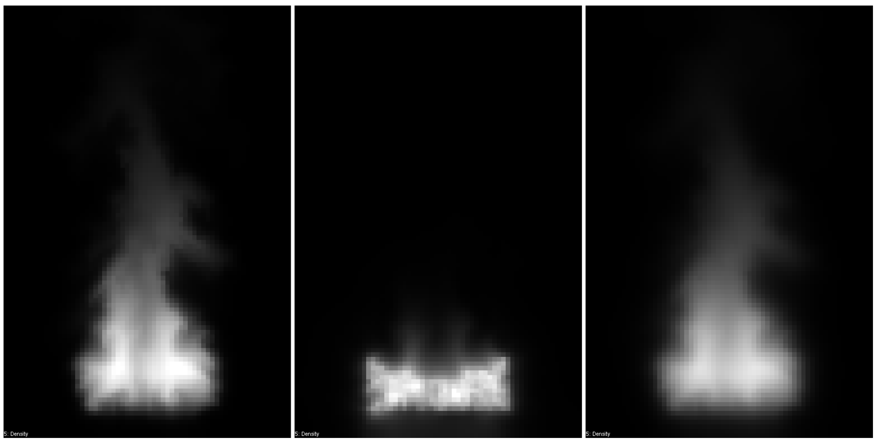

Figure 6.10: The exhaust gas (left), fuel gas (center), and temperature (right) parts of a 2D fire density texture.

compute these color textures at the end of the fire simulation, as they might be used by several visualization approaches.

The fragment program `calculateColors()` is used to compute the fire color texture. The exhaust gas and temperature values are fetched from the fire density texture and then used in combination with the black-body radiation texture to compute the fire color according to equation 5.2.3. The black-body radiation texture is precomputed by an instance of the class `BlackbodyRadiationTable` and uploaded to the GPU before the simulation starts. It is stored as a 2D texture with dimensions Nx1, where N is the size of the black-body radiation table.

The fragment program `calculateSmokeColors()` is used when computing the smoke texture. Again, the exhaust gas and temperature values are fetched from the fire density texture, and then the smoke density is calculated according to equation 5.3.1. We use part of a cosine function as the fall-off curve.

The alpha channels of the fire color texture and the smoke texture are used to store the fire color brightness and smoke amount respectively. These values are later used by the particle system implementation to determine whether the particles are below a user specified visibility threshold and should be respawned. In addition, the alpha channel of the smoke texture is used for smoke particle blending, whereas fire particle blending is done exclusively based on the RGB fire color.

## 6.5 Particle system

In this section we describe how the particle system presented in section 5.4 is implemented on the GPU. Our GPU implementation is based on the methods presented in [Lat04] and [KSW04], described in section 3.5. We use one particle system for the fire and one for the smoke, and the motion of both are controlled by the velocity texture from the fire simulation. The particle colors of the two particle systems are controlled by the fire color and smoke textures from the fire simulation respectively.

### 6.5.1 Overview

The main class used for particle system simulation and rendering is `ParticleSystem`. This class has two subclasses, `ParticleSystemSlice` and `ParticleSystem3d`, which are used in combination with the 2D slice and 3D simulations respectively. The `ParticleSystem` subclasses are initialized with the particle splat used for texturing, the particle count, and a `Flat3dTextureSet` instance in combination with the color and velocity texture identifiers from the `Fire3dSimulator` instance. The color texture is either the fire color or the smoke texture.

The particle system uses textures on the GPU to represent the state of the particles. The textures used are a particle position texture, a particle velocity texture, a spawn position texture, and a particle color texture. The textures all have the same dimensions, and although the textures are two-dimensional because of GPU size restrictions, they represent a one-dimensional array of particle data where each texel contains data for a specific particle. The RGB-channels of the particle position, spawn position, and particle velocity textures represent x, y, and z coordinates respectively. In addition, the alpha channel of the particle position texture is used to store the particle's spawn time. The RGBA-channels of the particle color texture naturally represent the particle colors.

The particle system simulation and visualization consist of a number of processes as shown in figure 6.11. First, the particle velocity texture is updated based on the velocity texture from the fire simulation. This velocity update is implemented differently in `ParticleSystemSlice` and `ParticleSystem3d`. Next, the particle velocity texture is used to update the particle position texture. Then, the particle color texture is updated using the fire color texture or smoke texture from the fire simulation. The color update is implemented similar to the velocity update and described in more detail later. Finally, the particle position and color textures are used when rendering the particles. The Cg programs used in the implementation of the particle system algorithm are encapsulated by `CgProgram` and `Texture2dOperation` instances from the GPU computational framework, which is utilized to perform the particle system simulation and rendering. The Cg programs are found in appendix D.4.

### 6.5.2 Particle domain

As the motion of the particles is controlled by the velocity field from the fire simulation, we define a particle domain that corresponds to the computational domain where the velocity field is defined. The particle positions are limited within this particle domain. In figure 6.12 the correlations between the computational domain of the fire simulation and the particle domain are shown.

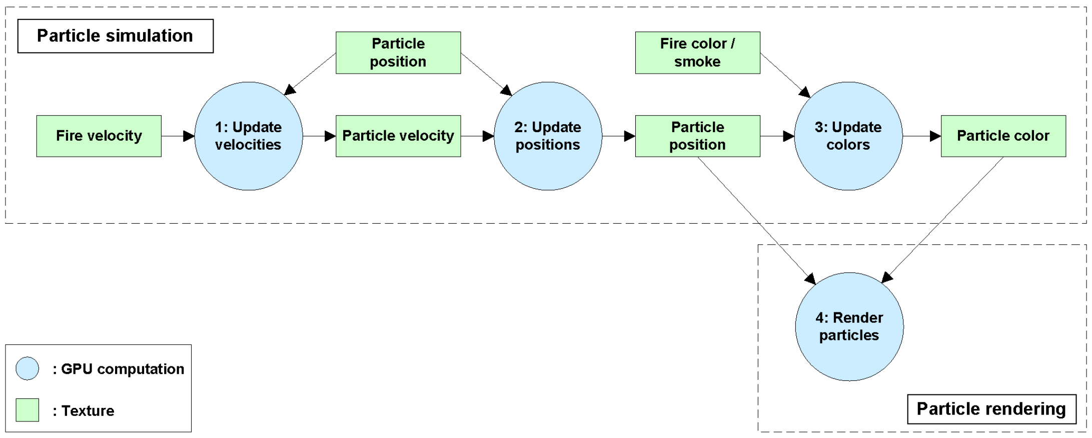

Figure 6.11: The processes involved in the GPU implementation of the particle system algorithm.

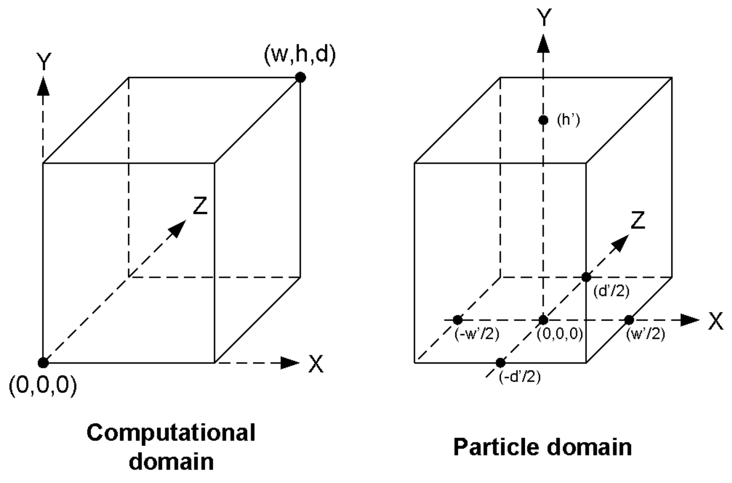

Figure 6.12: The correlations between the computational and particle domains.

Based on the width $w$, height $h$, and depth $d$ of the discretized computational domain, the width $w'$, height $h'$, and depth $d'$ of the particle domain are computed in the following way:

$$w' = 1 \tag{6.5.1}$$

$$h' = \frac{h}{w} \tag{6.5.2}$$

$$d' = \frac{d}{w} \tag{6.5.3}$$

### 6.5.3 Particle simulation

In this section we describe the three processes involved in the particle simulation part of the algorithm as shown in figure 6.11. The particle simulation is implemented in the member function `doRun()` of `ParticleSystem`.

#### 6.5.3.1 Update velocities

In order to guide the motion of the particles using a 3D fire simulation, the fragment program `setVelocities()` samples a particle's velocity from the velocity texture based on the particle position, and the results are written to the particle velocity texture. The particle position is fetched from the particle position texture and then used to perform a lookup into the flat 3D texture representing the velocity field using the Cg function `lookupValue()`. As the particle position is in the interval $[-\frac{w'}{2}, 0, -\frac{d'}{2}]$ to $[\frac{w'}{2}, h', \frac{d'}{2}]$, it must be transformed into the interval $[0, 0, 0]$ to $[w, h, d]$, where $w$, $h$, and $d$ are the computation width, height, and depth of the fire simulation. The transformed particle position is then used to fetch the velocity value from the velocity texture using trilinear interpolation. Finally, the resulting particle velocity is scaled to be valid in the interval $[-\frac{w'}{2}, 0, -\frac{d'}{2}]$ to $[\frac{w'}{2}, h', \frac{d'}{2}]$. This scaling ensures that the relative velocities of particles correspond to velocities in the velocity texture.

When a 2D fire slice simulation is used to guide the motion of the particles, `set-VelocitiesSlice()` is used instead of `setVelocities()` to sample the particle velocities from the velocity texture. The difference between the two fragment programs is the function used for fetching the particle velocity from the velocity texture, and `lookupVelocitySlice()` is a bit more complicated than `lookupValue()`. The particle position is transformed into the interval $[-\frac{w}{2}, 0, -\frac{w}{2}]$ to $[\frac{w}{2}, h, \frac{w}{2}]$, where $w$ and $h$ are the computation width and height of the tiles in the velocity texture respectively. The transformed particle position is then used to compute two indices specifying the two slices in the velocity texture that are closest to the particle position. Finally, the resulting particle velocity is computed using cylindrical interpolation between the two slices according to equation 4.4.2.

#### 6.5.3.2 Update positions

After the velocity for each particle has been determined and stored in the particle velocity texture, the velocity texture is in turn used by the fragment program `updatePo-sitions()` to update the position of each particle. The results from the fragment program are written to the particle position texture. The particle's current position and spawn position are fetched from the position texture and spawn position texture respectively. The particle position $x$ is then updated by performing an Euler step in the direction of the particle velocity $v$ using the following equation:

$$x' = x + v\Delta t \tag{6.5.4}$$

The particle position is reset to the spawn position if the particle has lived for more than $k$ seconds and its color alpha value is below a given threshold. The reason the particle has to live at least $k$ seconds before it is allowed to respawn is to make sure the particle has the opportunity to reach the location of fire. This is important as the spawn position might be slightly outside the fire location. When the alpha value of the particle drops below a given threshold it means that the particle no longer is located inside the fire, and can thus be respawned as it is nearly invisible.

#### 6.5.3.3 Update colors

In order to decide the color of each particle the fire color texture or the smoke texture are used for the fire particle system or the smoke particle system respectively. If the particle system is used to visualize the results from a 3D fire simulation, the fragment program `setColors()` is used to write the particle colors to the particle color texture. For each particle, the transformed particle position is used to fetch the particle color from the color texture using trilinear interpolation. The fragment program is almost identical to `setVelocities()`.

Another fragment program is used for setting the particle colors if a 2D fire slice simulation is used to control the particle system. `setColorsSlice()` uses another, more complicated method for fetching the particle color from the color texture. The fetching is achieved by `lookupSlice()` where the particle position is transformed into the interval $[-\frac{w}{2}, 0, -\frac{w}{2}]$ to $[\frac{w}{2}, h, \frac{w}{2}]$, where $w$ and $h$ are the computation width and height of the slices in the color texture respectively. As in `lookupVelocitySlice()`, the transformed particle position is used to compute two indices specifying the two slices in the color texture that are closest to the particle position. Finally, the resulting color value is computed using cylindrical interpolation between the two slices according to equation 4.4.1.

### 6.5.4 Particle rendering

Particles are rendered using viewplane-aligned quads centered at the particle positions. The quads are textured using a particle splat whose color is modulated by the particle's color. The particle rendering is implemented in the member function `renderParticles()` of `ParticleSystem`.

As particle positions are stored in textures, they need to be transferred to vertex coordinates in order to render the particles using OpenGL quads. We solve this by creating a vertex buffer whose vertex coordinates do not actually specify the position of the quads, but instead act as texture coordinates for the particle position and color textures and offsets for the quad corners. The xy and zw vertex components are used for the texture coordinates of the particle and quad corner offsets respectively. The vertex buffer is stored in a vertex buffer object on the GPU to avoid having to repeatedly transfer the vertex data each frame. Finally, the vertex buffer is transformed by the Cg vertex program `renderParticles()` and used to render textured quads to the framebuffer.

The Cg vertex program `renderParticles()` reads the particle position from the particle position texture and particle colors from the particle color texture, requiring hardware support for vertex shader texture fetches (Vertex Shader 3.0). The modelview projection matrix is used in combination with the particle size to position the offsets of the quad corners such that the quad lies parallel to the view plane. Finally, the offset is added to the fetched particle position and transformed by the modelview projection matrix.

In order to reduce fill-rate requirements, the particle splat used for texturing the particles should be cropped at the sides to avoid including areas of very low intensity. The result is that only an interior rectangle of the particle splat is used for texturing, requiring quad offsets and texture coordinates to be modified correspondingly. The cropping is performed automatically using a user-selected threshold value specifying the minimum particle splat texel intensity that should be visible. The interior rectangle of the particle splat is computed accordingly and the quad offsets and texture coordinates of the particle splat modified correspondingly.

## 6.6 Volume renderer

Volume rendering is performed by marching view rays through the fire volume, which is the volume containing the fire color field. Each ray has an entry point and an exit point, corresponding to the intersections of the ray with the bounding box of the fire volume. In this section we focus on how we implement the volume rendering algorithm presented in section 5.5.3.

The `RayMarcher` class is responsible for volume rendering, and the volume rendering algorithm is implemented in the member function `doRender()` of `RayMarcher`. The core of the volume renderer is implemented on the GPU for efficiency, utilizing Pixel Shader 3.0 functionality for dynamic looping. The fragment programs mentioned later in this section are encapsulated by `CgProgram` instances from the GPU computational framework, which is utilized to perform the ray marcher operations. The fragment programs can be seen in appendix D.5. In order to perform the ray marching, we render the front and back faces of the volume's bounding box, as done by [KW03] whose approach we described earlier in section 3.6.1.

### 6.6.1 Ray calculation

Initialization consists of computing a ray direction texture containing the normalized ray directions, later used during the ray marching. In addition, the alpha channel of this texture contains the distance from the entry points to the exit points. A temporary entry point texture is also used during initialization to store the entry points of the rays. Each view ray is thus represented by a corresponding texel in the two textures.

The dimensions of the two textures need to be the same as the dimensions of the main viewport. Still, only the parts of the textures that are covered by the bounding box of the volume are operated on. Figure 6.13 shows the bounding box of the fire volume and the RGB channels of the ray direction texture. The RGB channels are used for representing the x, y, and z components of the ray directions respectively.

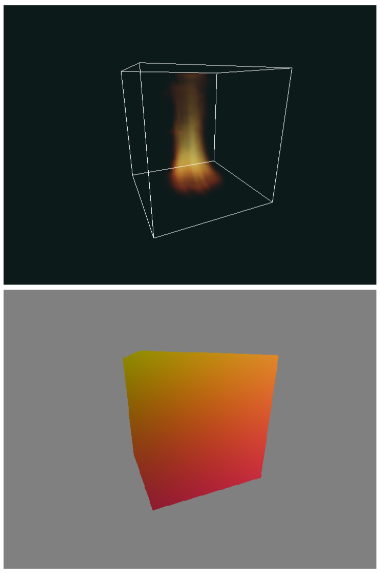
Figure 6.13: Top: volume bounding box, bottom: RGB channels of direction texture.

We first calculate the ray entry points by rendering the front faces of the bounding box using the fragment program `entryPointDetermination()`, writing the results to the entry point texture. When rendering the bounding box we set the texture coordinates at each corner to the position of that corner in global simulation coordinates, $[0, 0, 0]$ to $[w, h, d]$, where $w$, $h$, and $d$ are the width, height, and depth of the computational domain of the fire simulation. The fragment program used to compute ray entry points simply outputs the interpolated texture coordinates, resulting in a texture containing for each texel the entry point of the corresponding ray, or 0 if outside the projected bounding box.

Next, the ray directions and distances are calculated by rendering the back faces of the bounding box and using the fragment program `rayDirectionDetermination()`, writing the results to the ray direction texture. This fragment program gets the ray exit point from the interpolated texture coordinates and looks up the ray entry point from the entry point texture. It then subtracts the entry point from the exit point, resulting in an unnormalized direction vector. The length of this vector is returned in the alpha channel and the normalized direction vector is computed and returned in the RGB channels.

### 6.6.2 Ray marching

Ray marching is performed by again rendering the front faces of the bounding box to the framebuffer of the display and using the fragment program `rayMarcher()` or `rayMarcherSlice()` for 3D or 2D slice simulation respectively. The fragment program first gets the ray entry point from the interpolated texture coordinates and the ray direction from the ray direction texture. It then performs a loop, for each step incrementing a counter $t$ representing the current distance along the ray, as shown earlier in equations 5.5.1 and 5.5.2. An intensity value is accumulated by sampling from the fire color field using the Cg functions `lookup3d()` or `lookupSlice()` for 3D or 2D slice simulation respectively, until the loop is terminated when $t$ exceeds the distance from the entry point to the exit point, retrieved from the alpha channel of the ray direction texture.

After the loop has terminated, the accumulated color is returned from the fragment program. The resulting fragment is finally blended with the previous fragment of the framebuffer.

## 6.7 Dynamic illumination

We presented the dynamic illumination algorithm earlier in section 5.6.4. The implementation of the algorithm consists of the following classes: `Scene`, `SceneObject`, `Light`, and `Material`. The main class is `Scene`, which is responsible for managing lights and scene objects and rendering the scene. A scene object is any object which has a material and can be lit. `Scene` contains a set of `SceneObject` instances and a set of `Light` instances, representing scene objects and lights respectively. The `SceneObject` class in turn uses `Material` instances to represent surface properties. The Cg programs used in the implementation of the dynamic illumination algorithm are encapsulated by `CgProgram` instances from the GPU computational framework, which is utilized to perform the dynamic point light updates and per-fragment lighting. The Cg programs are shown in appendix D.6.

In the rest of this section we first discuss the handling and updating of lights in the scene, and then we present the implementation of the scene illumination algorithm.

### 6.7.1 Light initialization

In the implementation of the dynamic illumination algorithm we separate between three different kind of point lights: stationary static lights, stationary dynamic lights, and moving dynamic lights. A stationary static light has a fixed position and intensity and is used for global scene lights. The two types of dynamic lights get their intensities updated based on the fire color texture, and the moving dynamic lights additionally gets their positions updated based on the fire velocity texture.

In figure 6.14, the processes involved in the implementation of the light position and color updates are shown. The light positions are stored in a 1D light position texture where the RGB value of a texel corresponds to the x, y, and z positions of a light. The positions of the dynamic lights are defined in the same domain as the particles of the particle system, as shown earlier in figure 6.12. The base colors for each light are stored in a base light color texture, and the resulting colors are stored in a light color texture after scaling the base colors based on brightness values sampled from the fire color texture.

The positions and colors of the dynamic lights are stored in the leftmost part of respectively the light position texture, the light color texture, and the base light color texture. This enables us to operate only on the dynamic lights when updating the light color texture based on the fire color texture.

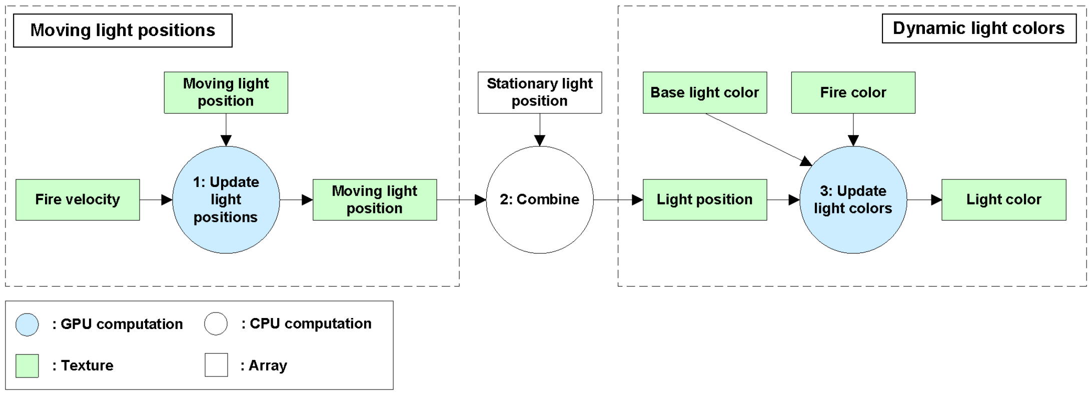

Figure 6.14: The processes involved in the implementation of the moving light position and dynamic light color updates.

#### 6.7.1.1 Dynamic light positions

The positions of the moving dynamic lights are controlled by an instance of `ParticleSystem3d` or `ParticleSystemSlice` for 3D or 2D slice simulation respectively. *Update light positions* of figure 6.14 thus corresponds to the first two processes of figure 6.11 for the particle system. The moving light position texture from the particle system is used to update the positions of the corresponding `Light` instances. Finally, the light position texture is filled with the positions of both moving and stationary `Light` instances in the member function `updateLights()` of `Scene`.

#### 6.7.1.2 Dynamic light colors

The colors of the dynamic lights are updated using the fragment program `sample-LightColors()` or `sampleLightColorsSlice()` for 3D or 2D slice simulation respectively. The fragment program is initiated from the member function `sampleLight-Colors()` of `Scene`, called from `updateLights()`. For the 3D case, the light positions are transformed into the interval $[0, 0, 0]$ to $[w, h, d]$, where $w$, $h$, and $d$ are the width, height, and depth of the discretized computational domain, whereas for the 2D slice case, the light positions are transformed into the interval $[-\frac{w}{2}, 0, -\frac{w}{2}]$ to $[\frac{w}{2}, h, \frac{w}{2}]$, where $w$ and $h$ are the width and height of the tiles in the fire color texture. Based on the transformed light position a color value is sampled from the fire color texture. The brightness of this color value is then used to scale the color value fetched from the base light color texture. Finally, the result of the fragment program is written to the light color texture.

### 6.7.2 Scene illumination algorithm

The scene illumination is implemented in the member function `render()` of `Scene` and is summarized by algorithm 1.

**Algorithm 1** Scene illumination algorithm

```
for all scene objects do
  transform camera position to object space for Phong lighting model
  for all light sources do
    transform light position to object space for Phong lighting model
  end for
  activate vertex and fragment programs for per-fragment lighting
  transform scene object to camera space
  render scene object
  deactivate vertex and fragment programs for per-fragment lighting
end for
```

The lighting model is local Phong illumination with no shadows, as described earlier in section 5.6.1. Each scene object has corresponding materials which specify the ambient, diffuse, and specular coefficients of the scene object's surface. The Phong illumination model requires, in addition to material parameters, the light positions and colors, the camera position, the surface position and normal, and the global ambient light of the scene. For each scene object, the light and camera positions are transformed into the local object space of the scene object. The scene object material parameters, the camera position, the light position and color textures, and the global ambient light are then set as parameters to the fragment program used for per-fragment lighting, `fragmentLightingFS()` and `fragmentLightingTexturedFS()` for textured and non-textured surfaces respectively. We used [FK03] as a guide for implementing the per-fragment lighting.

The vertex program used for per-fragment lighting, `fragmentLightingVS()`, simply transforms the vertices using the OpenGL modelview projection matrix. In addition, it passes through untransformed vertices, vertex normals, vertex colors, and texture coordinates to the fragment program. The interpolated untransformed vertex position and interpolated vertex normal are used as respectively the surface point and surface normal in the lighting model. The interpolated vertex color is the material color, which combined with the ambient, diffuse, and specular coefficients decides the surface properties. The material color is also multiplied by the texel fetched using the interpolated texture coordinates if texturing is used for the active scene object. Based on the surface properties, the fragment program first computes the global ambient contribution, before looping through all the light sources and computing their individual diffuse and specular contributions. The global ambient contribution and light contributions are accumulated and returned as the final fragment color.

## 6.8 Algorithm implementation

Earlier in this chapter we have presented all the modules that together constitute the complete algorithm implementation. In this section we look at how the complete algorithm is implemented on a per-frame basis, including an option of running the simulation at a fixed simulation timestep, decoupling the simulation frame rate from the frame rate of the application. For an overview of the application environment and simulation and visualization parameters, see appendix B.

### 6.8.1 Fixed simulation timestep

Instead of performing the physical simulation each frame, based on the timestep since the last frame, we have implemented an option of performing the simulation at a fixed timestep [Fie06]. Using a fixed timestep has several advantages:

- The simulation uses a predictable amount of computational resources. The same amount of time will be spent on simulation each second, whereas the rest can be spent on visualization and more.
- The simulation will behave exactly the same independent of the main frame rate. The quality of the simulation will not deteriorate under too low or too high frame rates.
- No more computational power than necessary is used to perform the simulation.

A fixed timestep of $\Delta T$ ensures that $\frac{1}{\Delta T}$ simulation steps are performed each second. To perform the simulation at the fixed timestep $\Delta T$, we accumulate the time since the last simulation timestep was performed in a variable $t_{acc}$. Then, a number of simulation steps are performed until $t_{acc} < \Delta T$. Algorithm 2 shows how this variable is used each frame to perform the simulation at a fixed timestep.

**Algorithm 2** Fixed timestep simulation

```
$t_{acc} = 0$
loop
  $\Delta t$ = time since last frame
  $t_{acc} = t_{acc} + \Delta t$
  while $\Delta T \leq t_{acc}$ do
    $t_{acc} = t_{acc} - \Delta T$
    perform simulation
  end while
  perform visualization
end loop
```

Optimally, linear interpolation should be performed between the last and current simulation steps, to ensure smooth simulation step transitions. We have not implemented this interpolation technique, as it would require a bit of redesign to our current framework.

### 6.8.2 Complete algorithm

In figure 6.15 we show how the main classes from the modules presented in this chapter are combined in order to constitute the complete fire simulation and visualization algorithm. The algorithm is implemented in the `run()` member function of the class `Engine`. An instance of the `Fire3dSimulator` class performs the fire simulation using a fuel source texture and black-body radiation texture as inputs. The results of the fire simulation used by the other classes are the velocity texture, fire color texture, and smoke texture. The velocity texture is used by all three instances of the `ParticleSystem` class to move the fire, smoke, and light particles. The fire color texture is used by the fire particle system to visualize the actual fire, while the smoke texture is used by the smoke particle system to visualize the smoke emerging from the fire. The light particle system is not visualized, as it only updates the positions of the moving lights and does not use a texture to give colors to the particles. An instance of `Scene` performs the dynamic illumination, using the moving light position texture and fire color texture to update the colors of both the moving and the stationary dynamic lights. The fire color texture is also used by an instance of `RayMarcher` when volume rendering the fire.

In algorithm 3 we give the sequence in which the complete algorithm is implemented with the class instances from figure 6.15. The given algorithm is concerned with both the simulation and visualization of one frame using a dynamic timestep. For the fixed timestep implementation, the simulation and visualization parts of algorithm 3 are substituted for the *perform simulation* and *perform visualization* steps of algorithm 2.

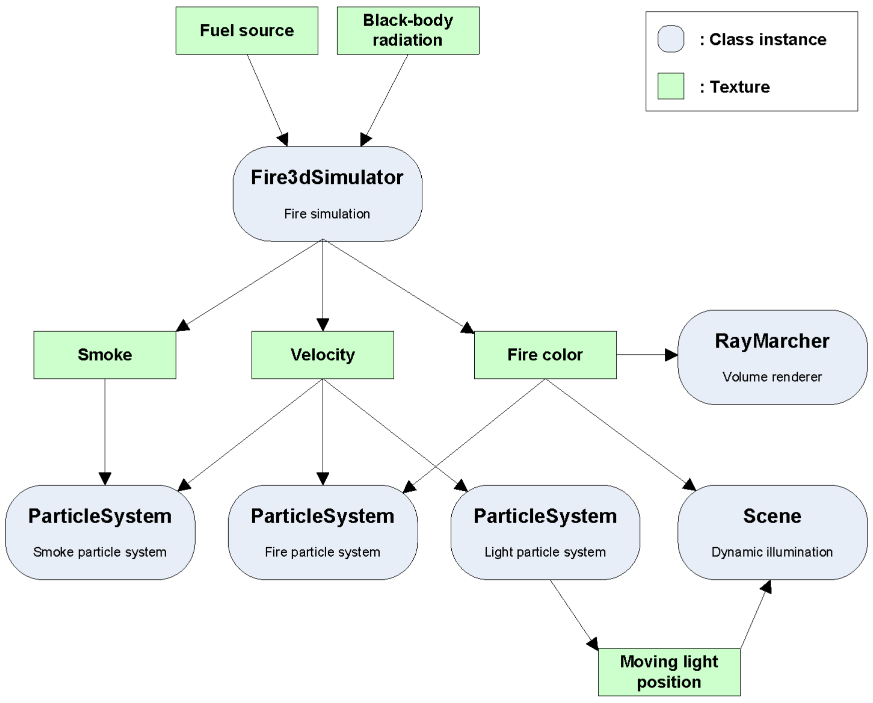

Figure 6.15: Composition of the complete simulation and visualization algorithm.

## 6.9 Extra features

We have implemented two extra features which were not part of the simulation and visualization algorithms, but which turned out to be very interesting. The two features are simulation domain advection and dynamic wind field respectively. The simulation domain advection allows a certain amount of interaction with the fire, in that the fire simulation behaves realistically as it is moved. This can for example be used in combination with a torch which can be moved by the user. When the torch is moved, the fire trails behind the torch as expected. Dynamic wind is a simple extension of the global wind boundary condition presented in section 4.3 to make the wind vary randomly over time, causing interesting motions of the fire.

### 6.9.1 Simulation domain advection

Our simulation domain advection method is an improvement of the method used by [WLMK02] to model fire movement. Whereas they use a less physically correct wind

**Algorithm 3** Complete simulation and visualization algorithm

```
loop
  $\Delta t$ = time since last frame
  Simulation:
  run fire simulation with timestep $\Delta t$
  if using fire particle system then
    run fire particle system with timestep $\Delta t$
  end if
  if using smoke particle system then
    run smoke particle system with timestep $\Delta t$
  end if
  if using light particle system then
    run light particle system with timestep $\Delta t$
  end if
  update scene lights
  Visualization:
  render scene objects with dynamic illumination
  if using particle systems then
    if using smoke particle system then
      render smoke particle system
    end if
    if using fire particle system then
      render fire particle system
    end if
  else if using volume renderer then
    render with ray marching
  end if
end loop
```

force at the boundary of the simulation domain proportional to the movement speed, we instead adjust the entire simulation domain to reflect the new position of the fire. A wind force at the boundary causes a small delay before the fire reacts as it is moved, whereas our method is instantaneous. In order to make the fire behave realistically as it moves, we advect both the particles in the particle systems and the simulation fields. The advection is based on a distance vector which specifies the distance the fire has been moved since the last simulation frame. When a fixed timestep is used, the distance vector is accumulated each frame until a new simulation step occurs.

Figure 6.16 shows a demonstration in 2D of how the simulation domain advection works. The left part of the figure shows the old (dashed lines) and new positions of the simulation domain. To perform the simulation domain advection, each cell of the new simulation domain is interpolated from the cells which it overlaps in the old simulation domain. The right part of the figure shows in green a sample cell and in blue the cells which are interpolated to get the new value. The simulation domain advection can be implemented by performing the same advection as in the fluid solver, but instead of a velocity field the distance vector is used for advection. Simulation domain advection does not work well with 2D fire slice simulation, thus we have only implemented it for 3D fire simulation by adding a uniform advection parameter to the fragment program `advect3d()` shown in appendix D.1. This parameter is set to the distance vector, enabling us to correctly update the velocity and fire density fields when the fire is moved.

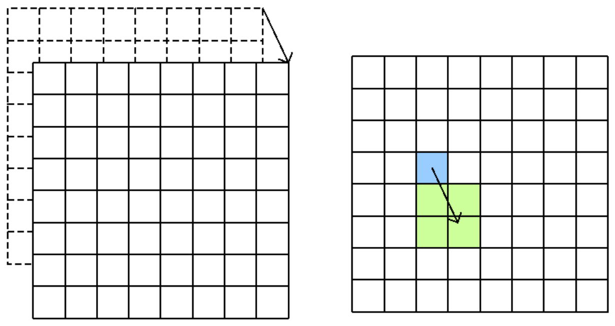

Figure 6.16: Advection of the simulation domain.

The particles in the particle systems must be advected correspondingly, to make sure that they do not become unsynchronized with the fire simulation domain. The particle system advection is done in the fragment program `advectParticles()` shown in appendix D.4, by simply adding the distance vector to each of the particle positions in the particle position texture.

### 6.9.2 Dynamic wind

Perlin noise was introduced in [Per85] and has among other been used in computer graphics to produce terrains, textures, procedural clouds, and other natural effects.

We use two one-dimensional Perlin noise curves to vary the wind vector used for the global wind boundary condition. The two curves are used to respectively sample the amplitude and the angle of the wind vector in the xz plane. We have not found it necessary to be able to have any wind in the y direction. The amplitude and angle curves are controlled with three parameters: a minimum value, a maximum value, and a frequency. The minimum and maximum values specify the range of values which the curve can take, whereas the frequency specifies how fast the curve changes. The following equation shows how the wind vector **w** is computed:

$$\mathbf{w} = \gamma \begin{pmatrix} \cos\alpha \\ 0 \\ \sin\alpha \end{pmatrix}, \tag{6.9.1}$$

where $\gamma$ is the wind amplitude and $\alpha$ is the wind angle. The wind amplitude and wind angle are calculated based on the Perlin curves and the current time $t$ as shown in equations 6.9.2 and 6.9.3. The *perlin* function is a time-varying Perlin noise curve with range [0, 1].

$$\gamma = \gamma_{min} + (\gamma_{max} - \gamma_{min}) \cdot perlin\,(t \cdot \gamma_{frequency}) \tag{6.9.2}$$

$$\alpha = \alpha_{min} + (\alpha_{max} - \alpha_{min}) \cdot perlin\,(t \cdot \alpha_{frequency}) \tag{6.9.3}$$

## 6.10 Tools and development environment

Table 6.4 shows the various tools we used for development. The implementation was mainly done in Microsoft Visual C++ .NET for application code, whereas the NVIDIA Cg Toolkit was used for Cg fragment and vertex programs. We used SDL and OpenGL for platform support (image loading, event handling, timing, etc) and rendering. We used GLEW for handling OpenGL extensions, and Aaron Lefohn's `FramebufferObject` class for rendering to textures. Finally, we utilized lib3ds to load 3D models, which we later use to showcase the dynamic illumination.

## 6.11 Summary

We have presented the implementation of the fire simulation algorithm and the fire visualization algorithms. This implementation is made possible by our GPU computational framework, which gives us easy access to the general-purpose computation aspects of the GPU. The framework is used by all the other modules of the implementation: fluid solver, fire simulation, particle system, volume renderer, and dynamic illumination. We have implemented a fluid solver for both 3D simulation and 2D slice

| Name | URL |
|---|---|
| MS Visual C++ .NET | `http://msdn.microsoft.com/visualc/` |
| NVIDIA Cg Toolkit | `http://developer.nvidia.com/object/cg_toolkit.html` |
| SDL | `http://www.libsdl.org/` |
| OpenGL | `http://www.sgi.com/products/software/opengl/` |
| GLEW | `http://glew.sourceforge.net/` |
| FramebufferObject | `http://sourceforge.net/projects/gpgpu/` |
| lib3ds | `http://lib3ds.sourceforge.net/` |

Table 6.4: Development tools.

simulation with volumetric extrusion. The fire simulation builds on the fluid solver to perform the main fire simulation. The velocity field, fuel gas field, exhaust gas field, and temperature field are all stored in flat 3D textures on the GPU, along with the fire color field and smoke color field used for visualization and other, temporary fields. The particle system uses a set of 2D textures to represent one-dimensional arrays of particle data, and the volume renderer uses textures to store ray entry points, ray exit points, and ray directions. Finally, for dynamic illumination the light positions and colors are stored in 1D textures used by a fragment program to perform per-pixel lighting. As we have shown, nearly all of the computations for both the fire simulation and visualization are performed on the GPU.

When performing the simulation a fixed timestep can be used to ensure stability and predictability. Using a fixed timestep, the simulation is performed a varying number of times per frame based on the frame rate. We have extended the simulation with simulation domain advection and dynamic wind, increasing the amount of fire effects we can model.

# Chapter 7

# Results

In this chapter we present the results from our implementation of the fire simulation and visualization algorithms presented in earlier chapters. First, we present and compare performance results by adjusting central parameters. Next, we present visual results[1] and discuss the visual quality of our fire using various simulation and visualization approaches. Finally, we compare our fire with footage from real fire and with previous approaches.

Table 7.1 shows the hardware specifications and setup of the machine which was used to produce all the results in this chapter.

| | |
|---|---|
| CPU | Dual Pentium 4, 3.0 GHz |
| RAM | 512 MB |
| GPU | Gainward GeForce 7800GT, 256 MB |
| GPU core clock | 450 MHz |
| GPU pixel pipelines | 24 |
| GPU texture units | 24 |
| Operating system | Microsoft Windows XP Professional Service Pack 2 |
| Graphics drivers | NVIDIA ForceWare 77.77 |

Table 7.1: Hardware specifications and setup.

## 7.1 Performance results

When obtaining the performance results, we used three different configuration setups and varied a selection of parameters while keeping the rest fixed. The detailed configuration setups are given in appendix C.2, and the accurate performance results with

[1]Screen captures are included in this chapter, whereas video captures can be found in the accompanying *media* folder.

parameter values used are shown in appendix C.1, which contains corresponding tables for each of the graphs and charts presented in the following sections. To get accurate results, we have not included any surrounding scene objects except from in the dynamic illumination and window dimensions results. All measurements of frame rates were calculated based on the average frame rate value of three runs, each lasting for 30 seconds. The frame rate for a run was calculated by dividing the time used for the run in seconds with the total number of frames in the run. To get accurate results, we performed all the program initialization before starting the timers.

### 7.1.1 Test parameters

Here we describe the effect of various parameters on the implementation of the fire simulation and visualization algorithms. The following set of parameters were varied when obtaining the performance results:

- *Simulation grid dimensions* - The dimensions of the discretized computational domain and whether the simulation is performed in 3D or a number of 2D slices with volumetric extrusion. Increasing the size of the simulation grid mainly affects the performance of the simulation part of the implementation.
- *Visualization* - Specifies whether the simulation results are visualized or not. This parameter is used when varying the simulation grid dimensions and when enabled we visualize the simulation results using a fire particle system with 2048 particles of size 0.04 and a smoke particle system with 512 particles of size 0.2.
- *Particle count* - The number of particles used in the fire or smoke particle systems. Increasing the number of particles affects the fill-rate requirements and thus the processing time needed for visualization.
- *Particle size* - The size of the particles used in the fire or smoke particle systems. The size of the particles also affects the fill-rate requirements and the processing time needed for visualization.
- *Particle rendering* - Specifies whether the particles of the fire or smoke particle systems are rendered to the framebuffer. By toggling this parameter, the time used for updating the particle positions, velocities, and colors can be compared to the time used for rendering the particles.
- *Volume rendering step size* - The step size used when tracing rays trough the fire color volume as part of the ray marching visualization method. Decreasing the step size leads to more sampling points along the trace rays and thus increases the processing time needed for visualization.
- *Point light count* - The number of point lights used for the dynamic illumination of the scene. The number of point lights greatly affects the processing time needed to illuminate the scene.

- *Window dimensions* - The size of the program window and thus the framebuffer in which the rendering occurs. Increasing the size of the program window increases the fill-rate requirements and thus the processing time needed for visualization and dynamic illumination.
- *Fixed timestep* - Specifies whether a fixed timestep is used when performing the fire simulation and the particle simulation of the fire and smoke particle systems. Using a fixed timestep makes sure that no more computational power than necessary is used to perform the simulation and that predictable simulation results are obtained. When active, a fixed timestep of $\frac{1}{30}$ s is used in the following sections.

### 7.1.2 Simulation grid dimensions

Figure 7.1 shows frame rates for 3D fire simulation only and 3D fire simulation with visualization at various simulation grid dimensions using a dynamic timestep. Fire and smoke particle systems were used when visualizing. The fire particle system had 2048 particles and particle size 0.04, and the smoke particle system had 512 particles and particle size 0.2. Both particle systems were defined in a particle system domain corresponding to the computational domain of the fire simulation. As can be seen in the figure, the frame rate rapidly decreases for the simulation only when the grid dimensions are increased. The frame rates for the simulation and visualization combined do not change that much, which shows that visualization is the bottleneck at low grid dimensions. When real-time frame rates are desired, grid sizes much larger than 32x48x32 can not be used. At this grid size the processing is quite evenly balanced between simulation and visualization. In figure 7.2 we also show the processing time per frame for the 3D fire simulation only as a function of the number of voxels in the grid. As expected, the processing time is linearly dependent of the number of voxels in the grid.

Figures 7.3 and 7.4 show frame rates for 2D fire slice simulation only and 2D fire slice simulation with visualization using a dynamic timestep. The same visualization approach as for 3D fire simulation was used. Frame rates are given for various slice dimensions and slice counts. As can be seen from figure 7.5, the number of slices has a near linear effect on the processing time for the slice simulation only results. When visualizing at low grid sizes the simulation is no longer the bottleneck, as is evident from the low frame rate variations at different slice counts in figure 7.4.

### 7.1.3 Particle systems

Figure 7.6 shows frame rates from the particle system visualization of the fire at various particle counts. A 2D fire slice simulation at 64x96 with 2 slices was used as the underlying simulation. Results are shown for dynamic timesteps (with and without rendering the particles) and fixed timesteps (with particle rendering). As can be seen from the

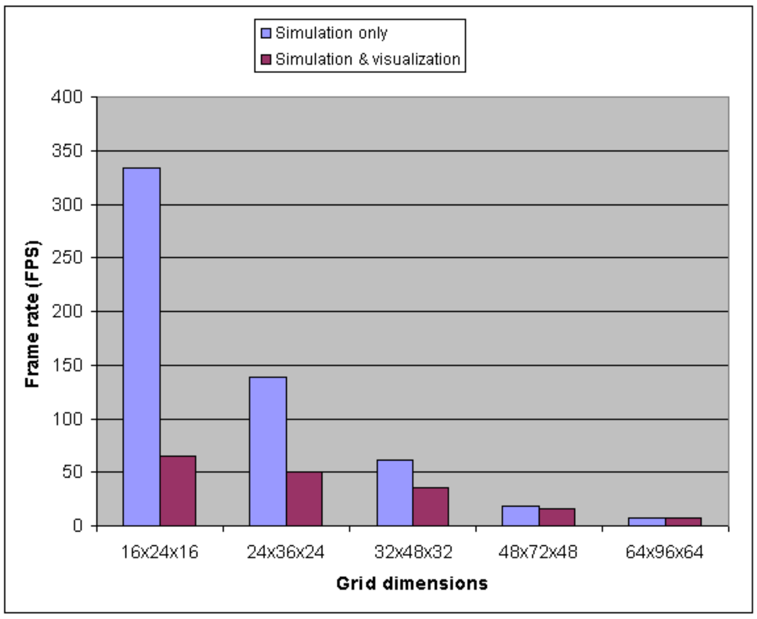

Figure 7.1: Frame rates for 3D fire simulation with and without particle system visualization at various simulation grid dimensions. (See table C.1)

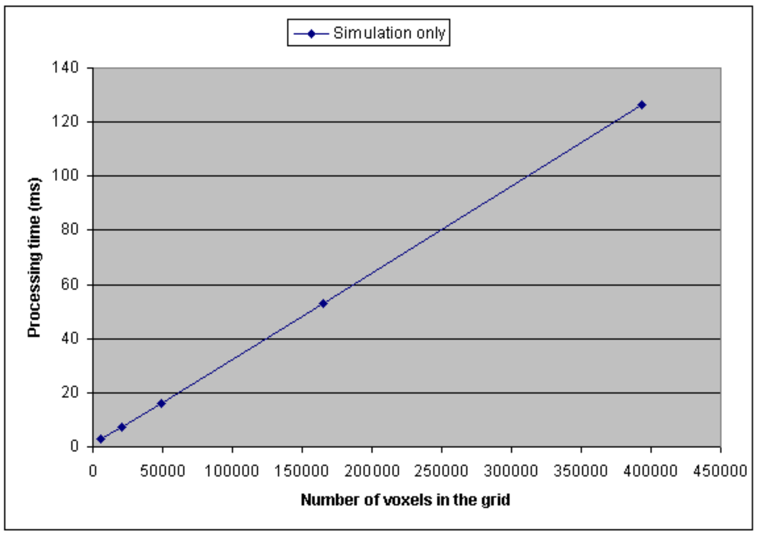

Figure 7.2: Processing time per frame for 3D fire simulation only as a function of the number of voxels in the grid. (See table C.1)

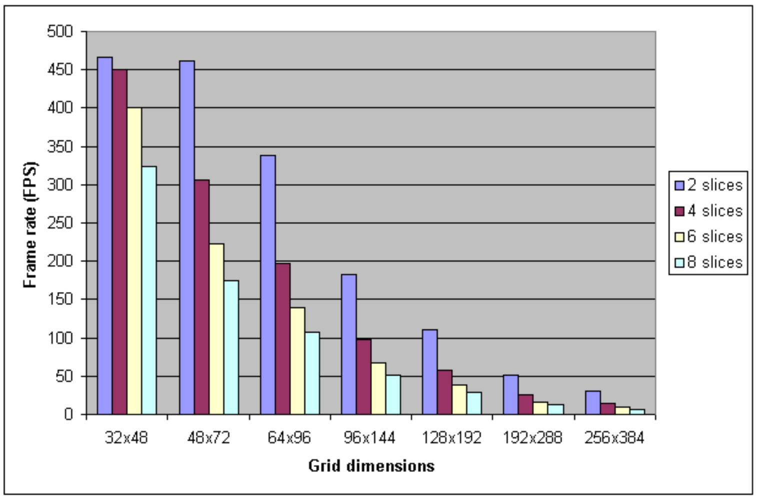

Figure 7.3: Frame rates for 2D fire slice simulation only at various simulation grid dimensions and slice counts. (See table C.2)

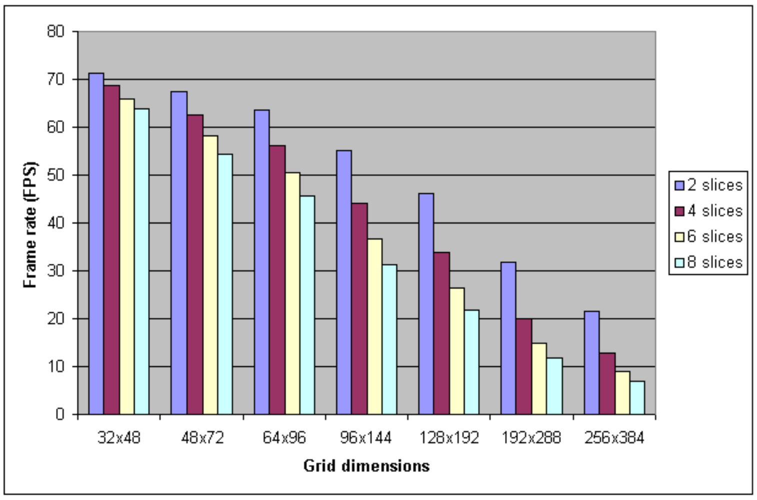

Figure 7.4: Frame rates for 2D fire slice simulation with particle system visualization at various simulation grid dimensions and slice counts. (See table C.3)

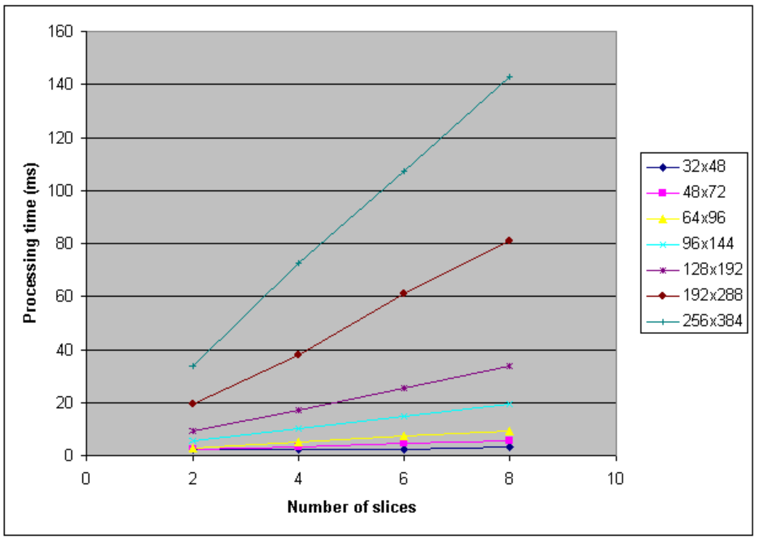

Figure 7.5: Processing time per frame for 2D fire slice simulation only at various simulation grid dimensions and slice counts. (See table C.2)

dynamic timestep results, there is not much overhead associated with increasing the particle count when the particles are not rendered and only the particle simulation is performed. This is because the fire simulation and not the particle simulation is the bottleneck. When particles are rendered however, due to high fill-rate requirements the particle rendering becomes the bottleneck. Due to adapting the particle size according to the particle count, we do not get a linear increase in processing time when increasing the particle count, as shown in figure 7.7. Thus real-time visualization can be achieved even for high particle counts.

Figure 7.8 shows the same results as figure 7.6 with the inclusion of the smoke particle system. We used the same underlying fire simulation as before, with a fire particle system of 2048 fire particles with size 0.04. By varying the smoke particle counts and sizes, the same trends as in figure 7.6 can be seen.

The results in figures 7.6 and 7.8 also show frame rates using a fixed timestep of $\frac{1}{30}$ s, which implies a simulation frame rate of 30 FPS. This means that when the frame rate is higher than 30 FPS, fixed timestep will produce higher frame rates than dynamic timesteps, as the simulation is not run each frame. On the other hand, when the frame rate is lower than 30 FPS, the simulation must sometimes run more than once per frame to catch up, which causes fixed timestep to produce lower frame rates than dynamic timesteps. In figures 7.6 and 7.8 we clearly see that the fixed timestep produces higher frame rates than the dynamic timestep.

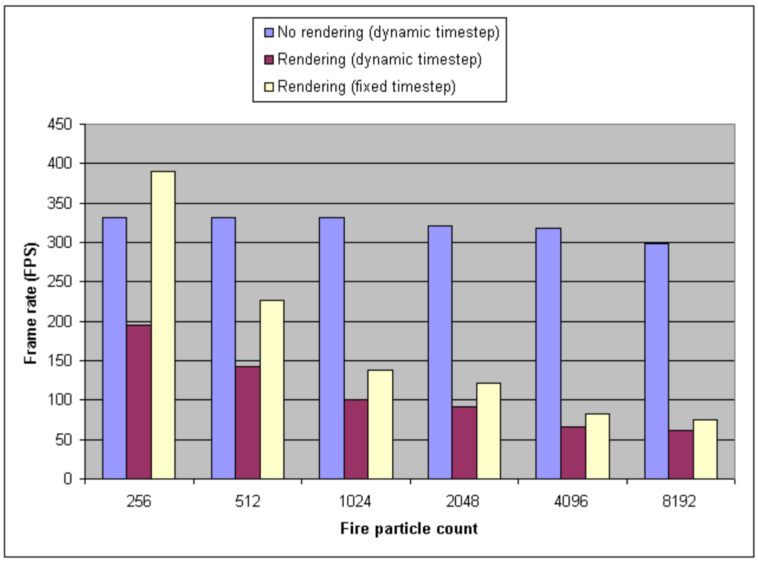

Figure 7.6: Frame rates for particle system visualization of fire at various fire particle counts. (See table C.4)

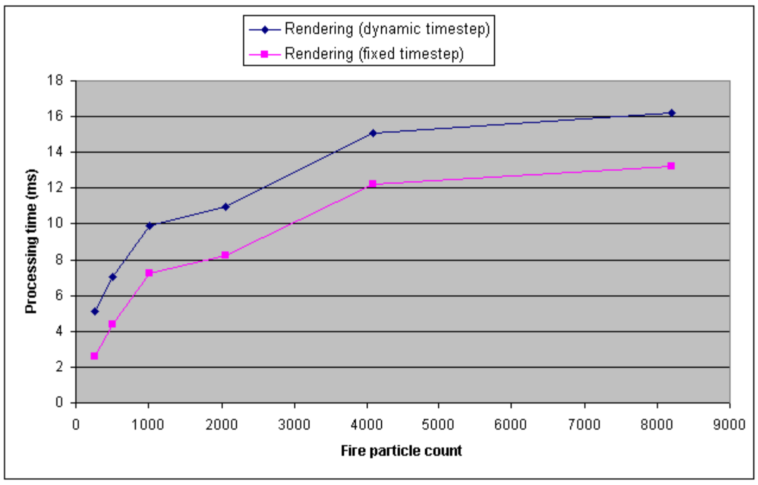

Figure 7.7: Processing time per frame for particle system visualization of fire at various fire particle counts. (See table C.4)

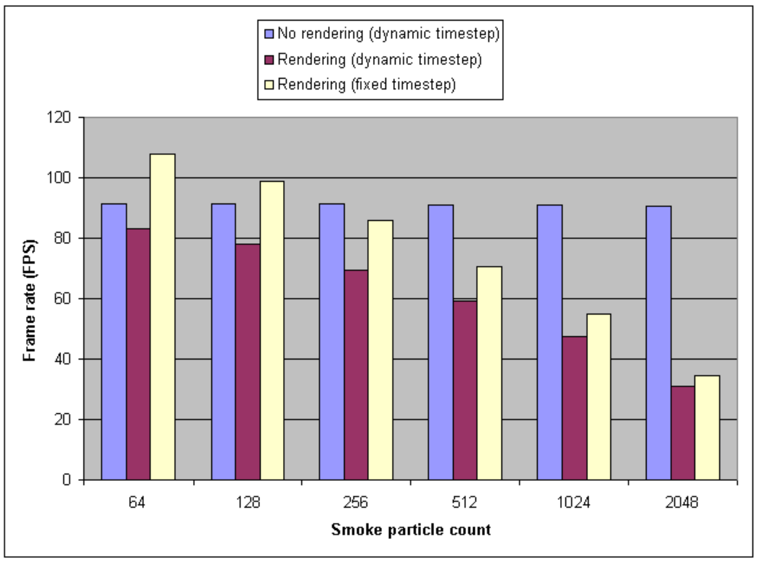

Figure 7.8: Frame rates for particle system visualization of fire and smoke at various smoke particle counts. (See table C.5)

### 7.1.4 Volume rendering

Figure 7.9 shows frame rates from the volume rendering of the fire using an underlying 3D fire simulation at various grid dimensions. The step size of the ray marching was 1.0 for all the results. As can be seen, only small grids can be visualized in real-time using volume rendering. As the frame rates are rather low, the results using a fixed timestep are consistently lower than when using a dynamic timestep.

Figure 7.10 shows frame rates from the volume rendering of the fire using an underlying 3D fire simulation at grid dimensions 24x36x24 with varying step sizes. The results show that the frame rates are highly dependent on the step size used. However, too large step sizes cause less accurate visualization or even aliasing artifacts. The frame rate when visualizing the fire using volume rendering is also highly dependent on the viewport dimensions and the part of the viewport covered by the bounding box of the fire. As a step size of 1.0 or less should be used to produce satisfying results, volume rendering is not quite suitable for real-time visualization of fire effects which cover large parts of the viewport.

### 7.1.5 Dynamic illumination

Figures 7.11 and 7.12 show frame rates when dynamic illumination is activated. The same fire simulation as for the particle system results in section 7.1.3 was used, and the fire was visualized with the fire and smoke particle systems just like for the simulation grid results in section 7.1.2. In addition, a couple of scene objects were included, such

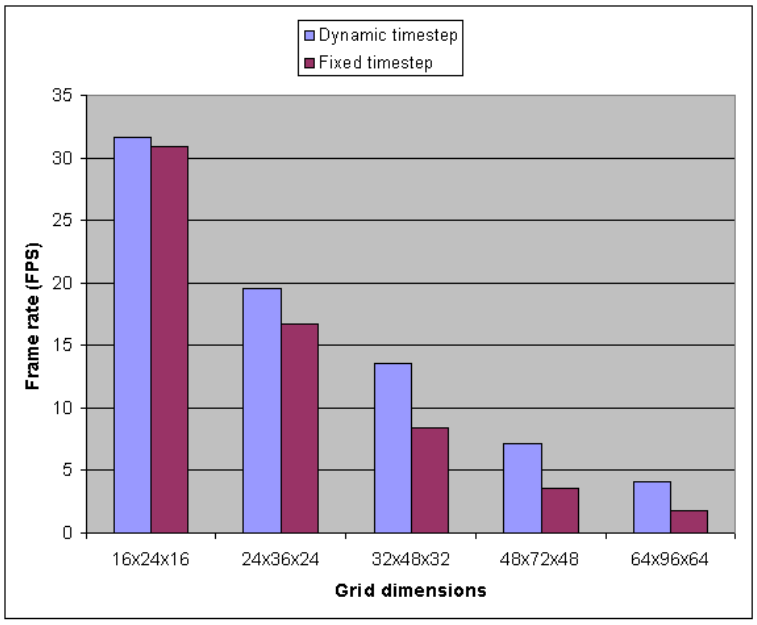

Figure 7.9: Frame rates for volume rendering the fire with no smoke using a 3D fire simulation at various grid dimensions. (See table C.6)

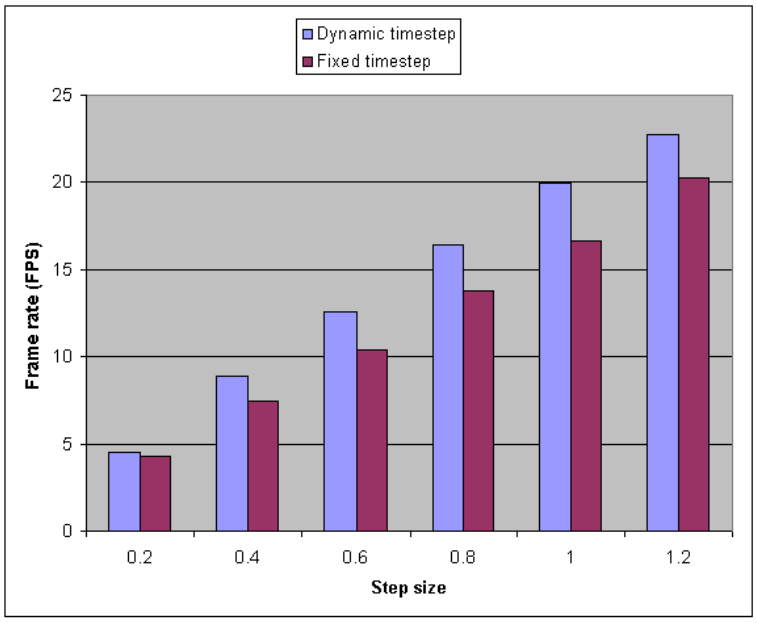

Figure 7.10: Frame rates for volume rendering the fire with no smoke at various step sizes using a 3D fire simulation at 24x36x24. (See table C.7)

that per-fragment lighting would be performed as part of the dynamic illumination.

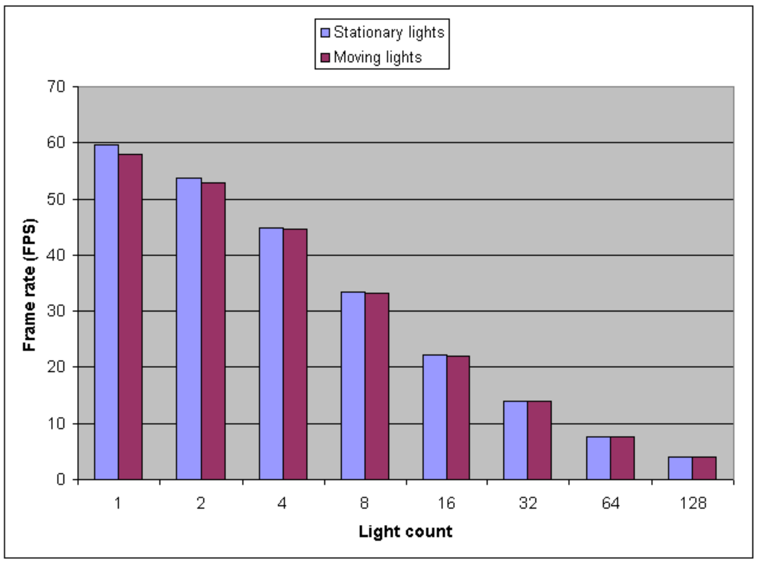

Figure 7.11: Frame rates for dynamic illumination with stationary and moving dynamic lights at various light counts using a dynamic timestep. (See table C.8)

As can be seen from the results, the frame rate decreases steadily as the number of lights are increased. From figure 7.13 we see that the processing time increases linearly when the light count is increased. This is because the per-fragment lighting needs to loop through each light to compute its contribution to the given fragment. The frame rate is only slightly lower when moving lights are used, which shows that the illumination and not the light particle system is the bottleneck. The number of lights that can be used when still achieving a real-time visualization lies somewhere between 8 and 16. The results also show that a fixed timestep causes the frame rate to increase at high frame rates and decrease at low frame rates with respect to the frame rates using a dynamic timestep, as we mentioned earlier in section 7.1.3.

### 7.1.6 Window dimensions

Figure 7.14 shows how the window dimensions affect the frame rate. The same simulation and visualization setup as in section 7.1.5 was used with a fixed timestep, in addition to dynamic illumination from four stationary dynamic lights. As can be expected, the frame rate decreases as the window dimensions increase. This is mainly due to the increased fill-rate of the particle rendering and the increased number of fragments that must be dynamically lit when rendering the scene.

Figure 7.15 uses the same simulation setup as before, but the simulation is run at a dynamic timestep and no visualization is used. As we use double buffering, we have to swap the off-screen buffer and the framebuffer at the end of each frame. The

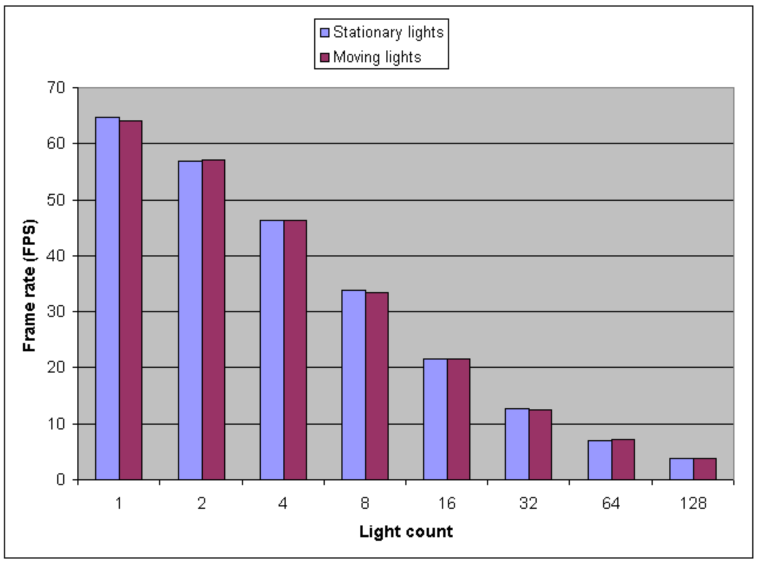

Figure 7.12: Frame rates for dynamic illumination with stationary and moving dynamic lights at various light counts using a fixed timestep. (See table C.9)

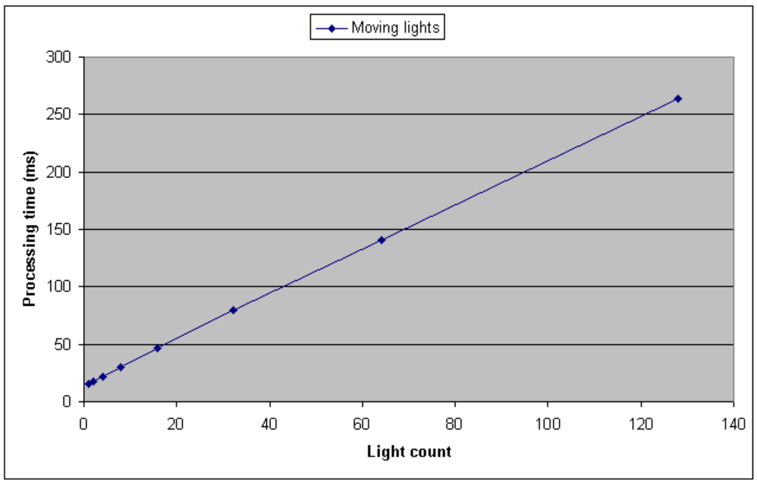

Figure 7.13: Processing time per frame for dynamic illumination with moving dynamic lights at various light counts using a fixed timestep. (See table C.9)

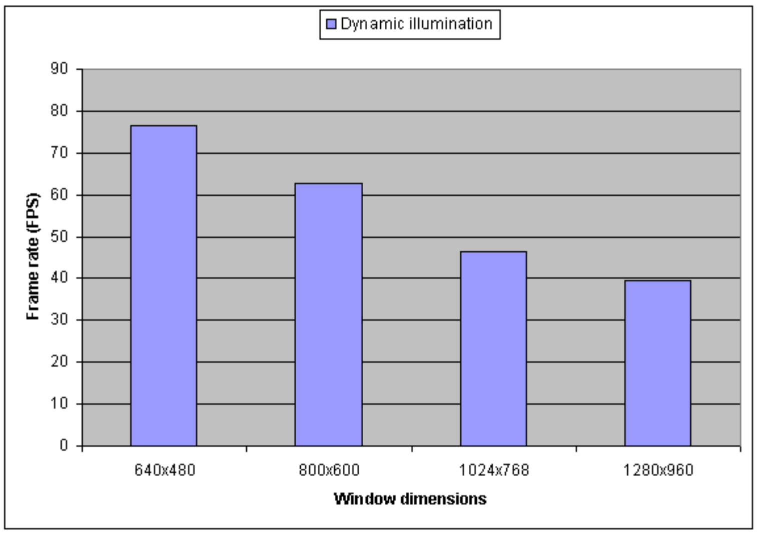

Figure 7.14: Frame rates for simulation and visualization of fire and smoke with dynamic illumination at various window dimensions. (See table C.10)

figure shows the results both when flipping buffers by calling `SDL_GL_SwapBuffers()` and when not flipping buffers. As can be seen, when flipping buffers there is a slight decrease in frame rate at larger window dimensions. However, when no visualization is performed there is no need to flip buffers, as the flipping has no affect on the simulation. The results when not flipping show that the performance of the simulation is independent of the window dimensions, which is logical as the simulation is performed in textures whose dimensions are independent of the window dimensions.

## 7.2 Visual results

In this section we look at the visual results achieved using various configurations. Although we only present still images, we will mean both the motion and appearance of the fire and smoke when we discuss visual quality. All the visual results in this section are supplemented with frame rates. We use two scenes to present the visual results: a campfire scene and a torch scene. When not explicitly specified, the following settings have been used:

- Window dimensions of 1024x768.
- For 3D fire simulation, a grid size of 24x36x24.
- For 2D fire slice simulation, a grid size of 64x96 with two slices.
- A fixed timestep of $\frac{1}{30}$ s.

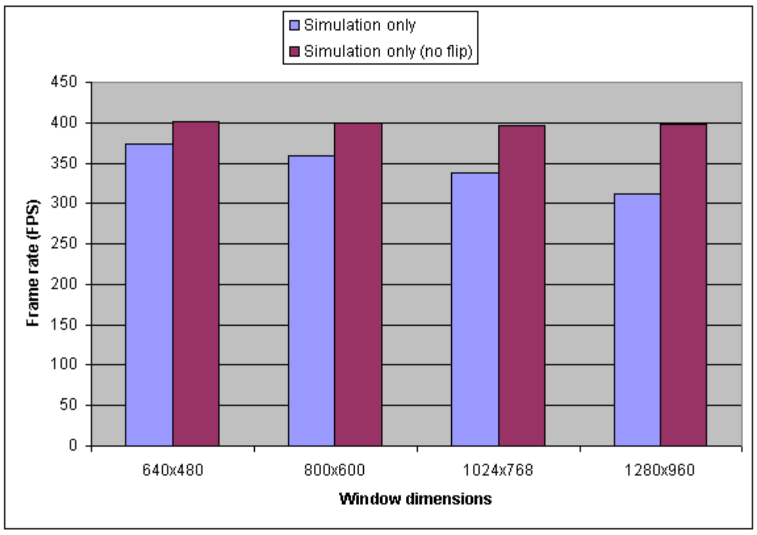

Figure 7.15: Frame rates for simulation only at various window dimensions. The *no flip* column corresponds to a modified version of the program were SDL_GL_SwapBuffers() was deactivated. (See table C.11)

- For the fire particle system, a particle count of 2048 and a particle size of 0.04.
- For the smoke particle system, a particle count of 512 and a particle size of 0.2.
- For volume rendering, a step size of 1.0.
- For the campfire scene, three stationary dynamic point lights and one static point light. The dynamic point lights were positioned along the central axis of the fire simulation.
- For the torch scene, two stationary dynamic point lights and one static point light.
- The torch scene always uses the underlying 3D fire simulation and the fire particle system (no smoke).
- When not using a global wind boundary, the rough boundary condition was used.

### 7.2.1 Simulation grid dimensions

The visual results from the 3D fire simulation at various grid dimensions are shown in figure 7.16. When using small simulation grids, much of the low-level turbulence is lost and the motion of the particles becomes somewhat uniform and smooth as particles close to each other sample their velocities from the same grid cells. Still, high-level turbulence is present, causing the fire to sway from side to side. Much of this high-level turbulence important for achieving realistic flame motion is lost when using large simulation grids, but instead more low-level turbulence is introduced. When using a simulation grid of 24x36x24 we achieve suitable amounts of both low-level and high-level turbulence.

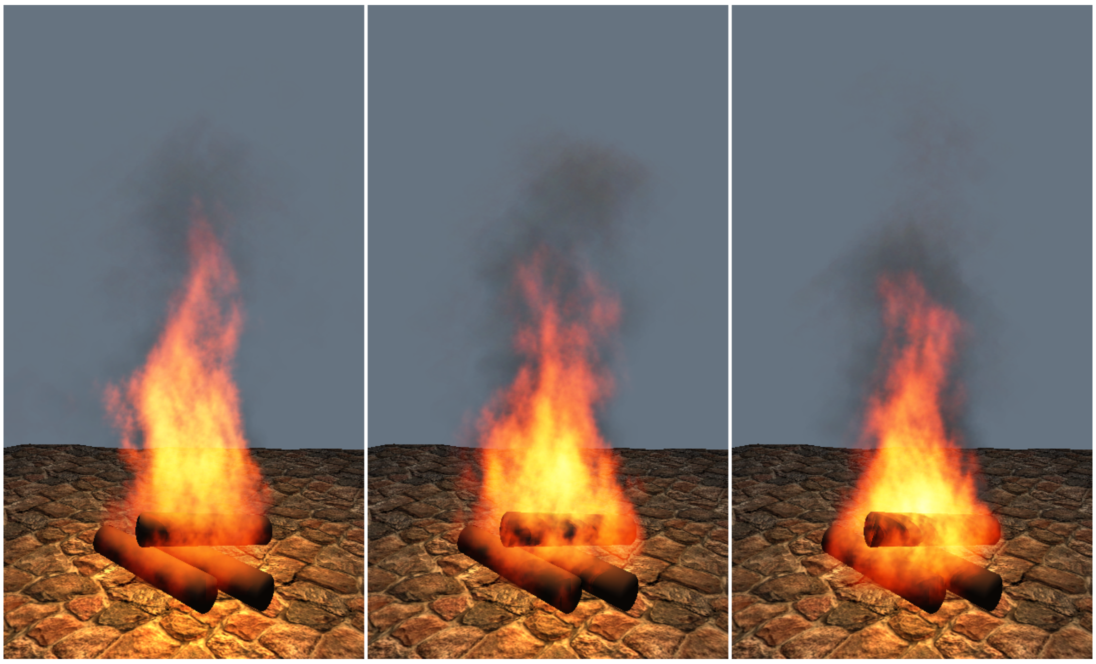
Figure 7.16: 3D fire simulation at various grid dimensions. Left: 16x24x16 @ 49 fps, center: 24x36x24 @ 42 fps, right: 32x48x32 @ 29 fps.

Figure 7.17 shows visual results from the 2D fire slice simulation with two slices at various grid dimensions. When the fire simulation parameters and fuel source field have been adjusted for a specific grid size it is a bit hard to achieve similar results at different grid sizes. Still, there is definitely a lower limit below which it is infeasible to achieve satisfying visual results. We achieve good results at moderately sized grids like 64x96, and have not experienced any improvements when increasing the grid size above this. In fact, at large grid sizes we seem to lose some of the high-level turbulence which is important for achieving realistic flame motion. At too small grid sizes the motion of the fire also suffers, becoming erratic and out of control.

Figure 7.18 shows visual results from the 2D fire slice simulation with grid dimensions 64x96 at various slice counts. The difference in visual quality between the different slice counts is rather low. The underlying fire simulation is less symmetric at higher slice counts, but this does not matter much for the visualization. The number of slices is more important when the fire becomes less rotationally uniform due to wind. Otherwise, even one slice can be acceptable, but as the performance drop from one to two slices is rather insignificant, there is usually no reason not to use at least two slices.

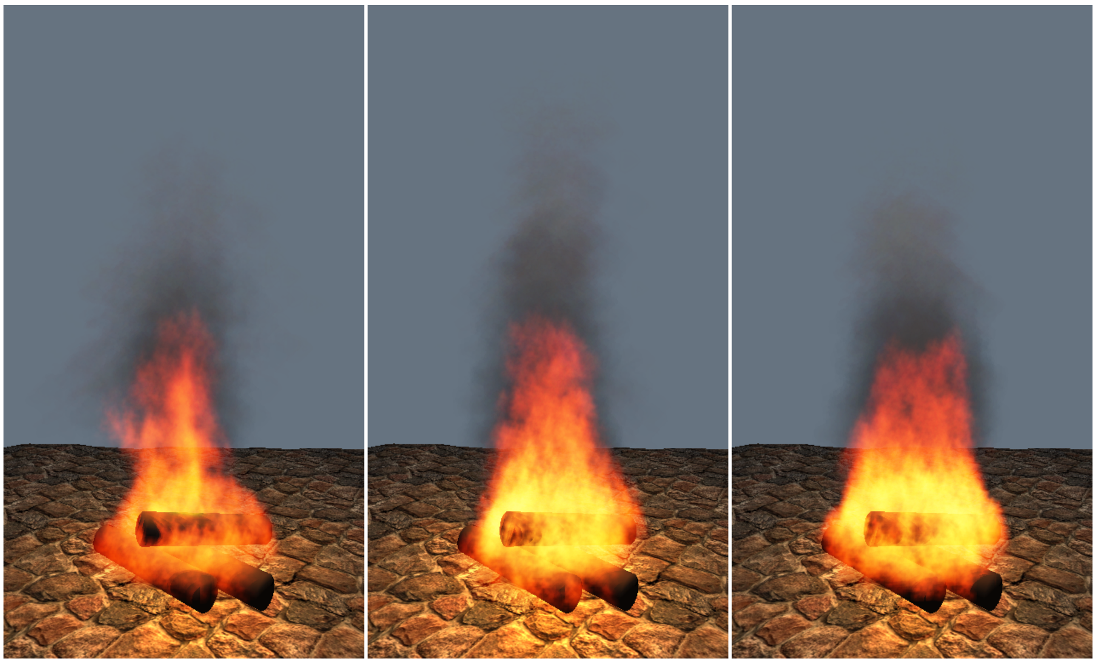

Figure 7.17: 2D fire slice simulation with two slices. Left: 32x48 @ 52 fps, center: 64x96 @ 49 fps, right: 128x192 @ 39 fps.

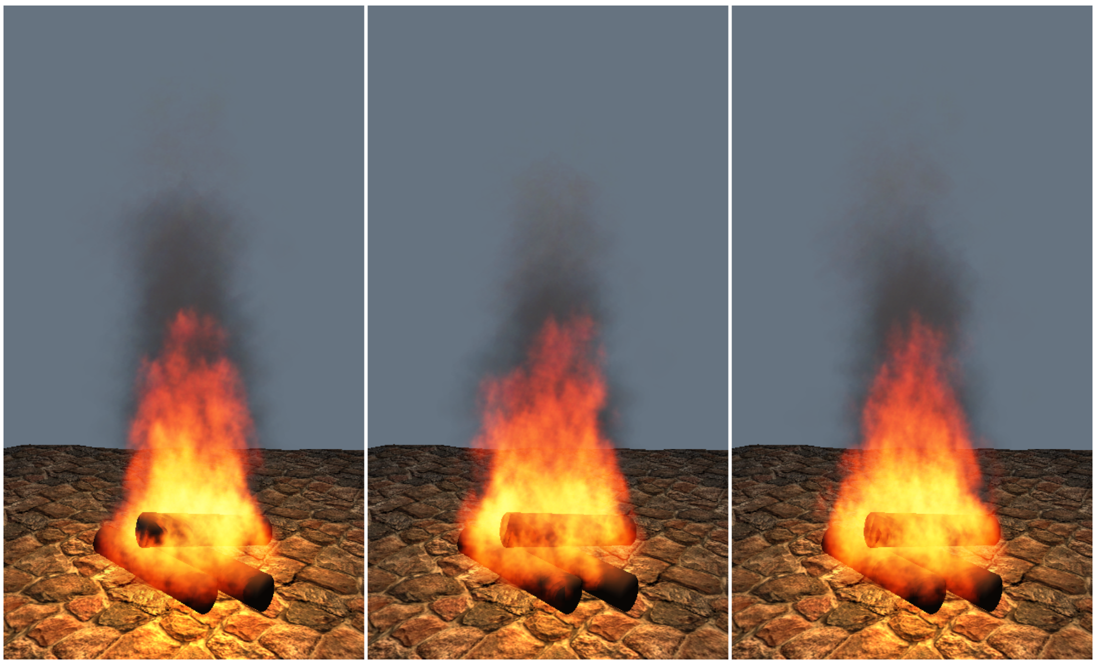

Figure 7.18: 2D fire slice simulation at 64x96. Left: One slice @ 51 fps, center: two slices @ 49 fps, right: eight slices @ 39 fps.

### 7.2.2 Particle counts

In figure 7.19 we show the fire visualized with particle systems of varying particle counts with the underlying 3D fire simulation. The particle sizes have also been adjusted to suit the particle counts. As can be seen, the leftmost fire is very smooth compared to the rightmost fire. However, too many particles causes the low-level texture detail due to the particle textures to be lost, in addition to a high reduction in frame rate. Depending on the requirements a low particle count can be used, for example if the fire will only be viewed from a distance. The frame rate gain when viewing the fire from a distance is also substantial. It would be possible to use a level-of-detail approach to adjust the particle count and particle sizes based on the distance from the fire.

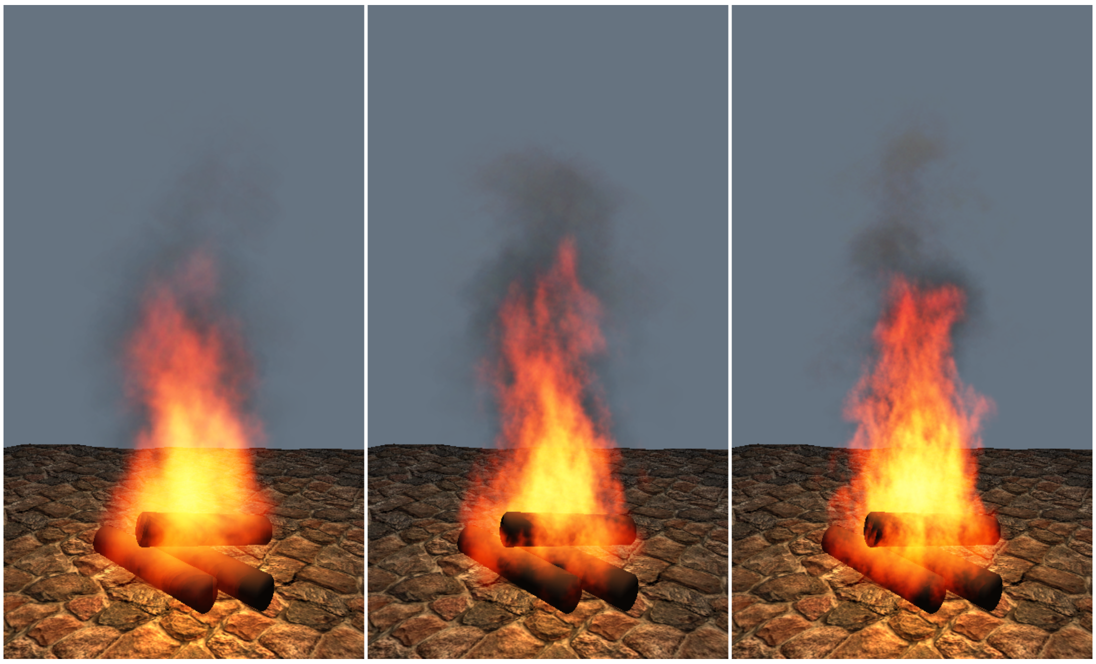
Figure 7.19: Various particle counts using the 3D fire simulation. Left: 512 fire particles and 128 smoke particles @ 65 fps, center: 2048 fire particles and 512 smoke particles @ 42 fps, right: 8192 fire particles and 2048 smoke particles @ 20 fps.

### 7.2.3 Smoke

Figure 7.20 shows the fire visualized with and without smoke with the underlying 2D fire slice simulation. Even though we use 2048 particles for the fire and only 512 particles for the smoke, there is a significant drop in frame rate when the smoke is activated, as the fill-rate requirements increase due to the larger particle size of the smoke particles. As can be seen, the smoke greatly increases the realism of the visualization. As the smoke particle system is visualized first, it also acts as a backdrop for the fire. Without the smoke, the fire looks too transparent when a bright background is used.

This is due to the fact that the fire is mainly emissive, whereas the smoke absorbs more of the background light.

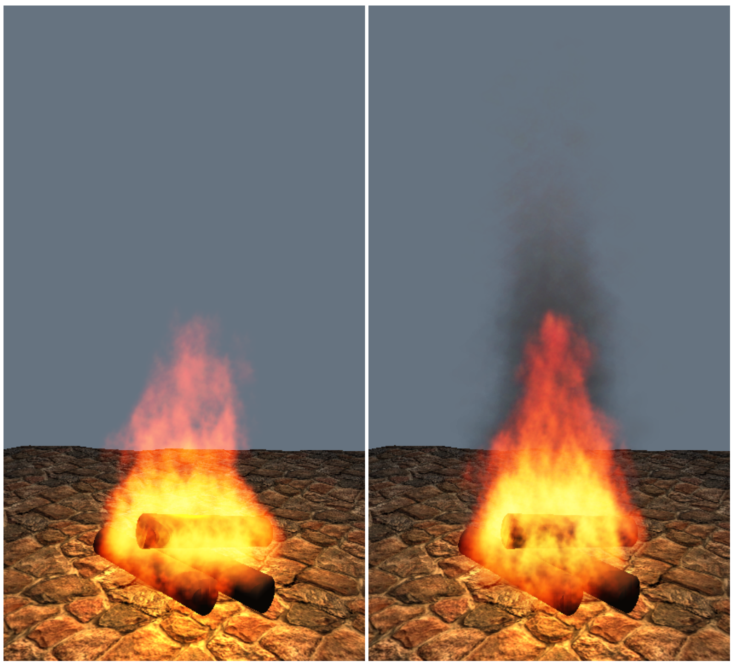

Figure 7.20: Smoke demonstration using the 2D fire slice simulation. Left: No smoke @ 69 fps, right: Fire and smoke @ 49 fps.

### 7.2.4 Simulation domain advection

We presented simulation domain advection earlier in section 6.9.1, and the effect of using this advection is shown in figure 7.21. Both the 3D fire simulation and the fire particle system are advected according to the motion of the torch, which we move from the left to the right. The fire trails realistically behind the torch as it is moved and returns to a natural position as the motion of the torch stops.

### 7.2.5 Global wind

In figure 7.22 we show the visual results when using the global wind boundary condition with the 3D fire simulation. As can be seen in the rightmost image, the fire is swaying realistically to the right due to the global wind coming from the left. An unfortunate side effect is that much of the illumination from the fire is lost as the three stationary dynamic point lights used are positioned along the central axis of the fire. When the fire sways to the right these point lights get a much lower intensity than the point lights used with the fire in the leftmost image. This side effect can be avoided by using

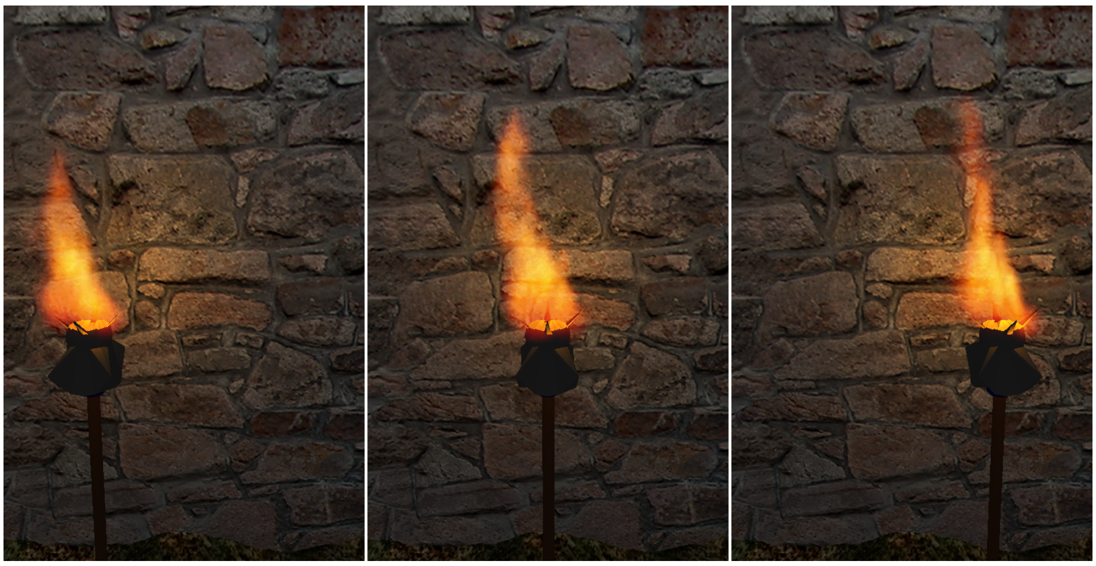

Figure 7.21: Three timesteps of the 3D simulation with the fire particle system, both advected according to the motion of the torch @ 29 fps.

moving lights instead, as shown in the next section. The results also show that the frame rate is not affected when using the global wind boundary condition.

Figure 7.23 shows the visual results of the global wind effect in combination with the 2D fire slice simulation. When using only two slices and a global wind vector pointing to the right the fire looks very unnatural. The slice positioned perpendicular to the global wind vector is not affected at all, making the fire split into two parts. This effect shows that volumetric extrusion is not very suitable when the fire simulated no longer has relative axial symmetry. Using more slices partially alleviates this problem, but as can be seen in the rightmost image not all parts of the fire sway to the right even when using eight slices. Thus, it is always preferable to use a 3D fire simulation when global wind is wanted.

In figure 7.24 we show the effect of using a global wind vector that is dynamically updated. The amplitude and direction of the vector are controlled by two Perlin noise curves, and this results in a fire that sways from side to side as if it was located outside, where the wind can be rather chaotic and unpredictable.

### 7.2.6 Dynamic illumination

In figure 7.25 the effects of using dynamic point lights can be seen. The leftmost image has only a static light and looks rather bland next to the rightmost image, which in addition to the static light has three stationary dynamic lights positioned along the central axis of the fire. The dynamic illumination does a great deal to enhance the visual quality of the scene in combination with the fire. With dynamic lights, the light

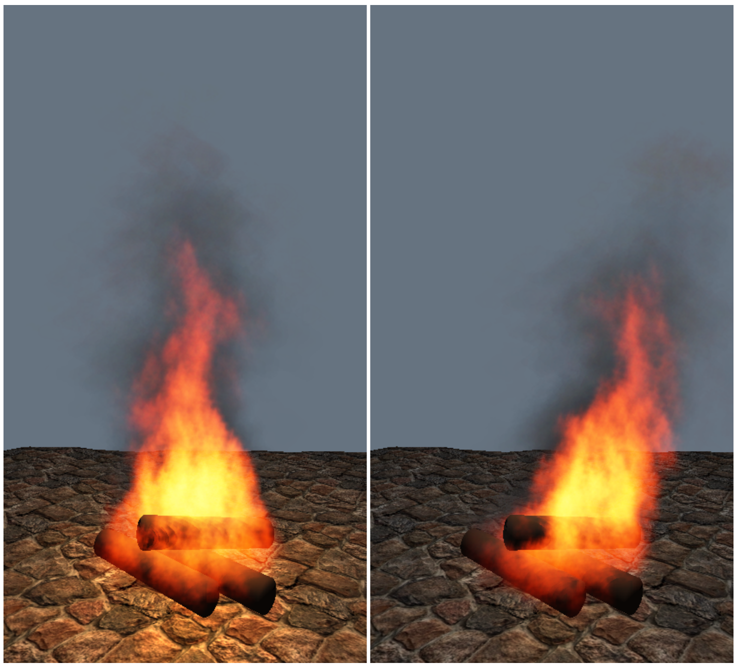
Figure 7.22: The 3D fire simulation. Left: No wind @ 42 fps, right: Global wind from the left @ 42 fps.

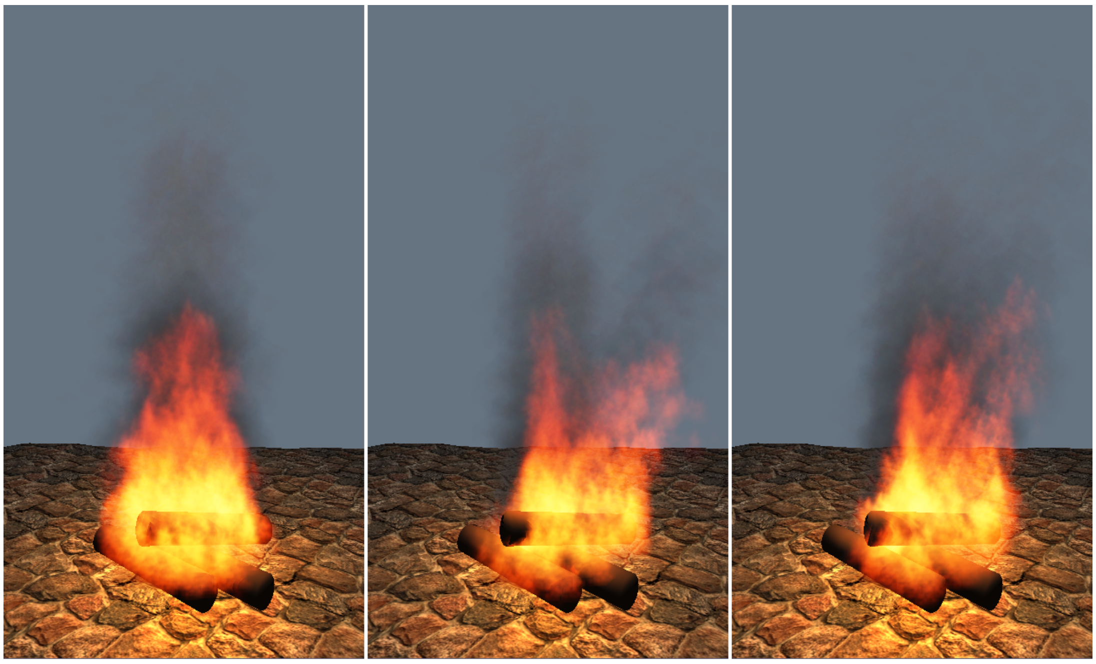
Figure 7.23: 2D fire slice simulation at 64x96. Left: 2 slices and no wind @ 49 fps, center: 2 slices and global wind from the left @ 49 fps, right: 8 slices and global wind from the left @ 39 fps.

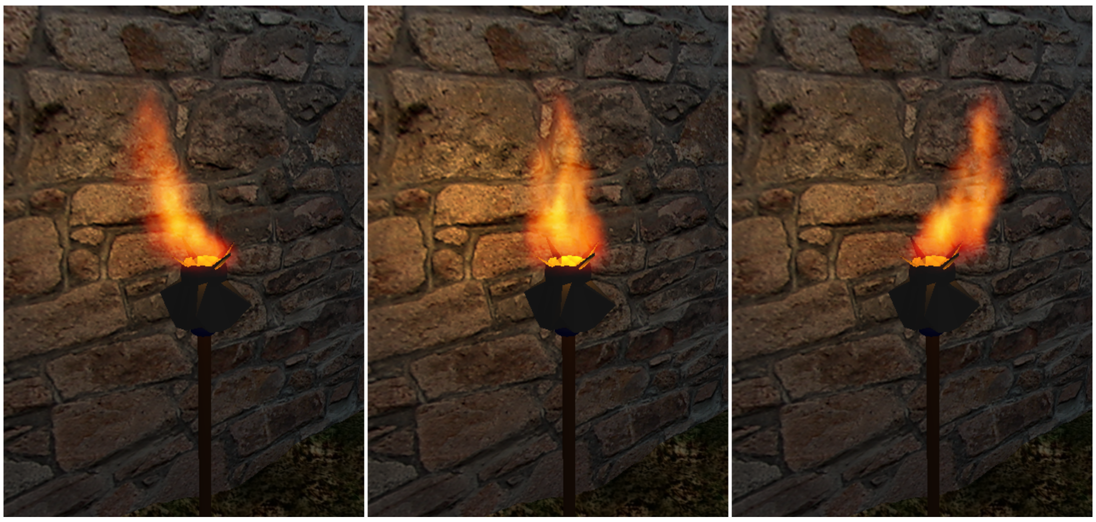

Figure 7.24: Three timesteps of the 3D simulation of a torch affected by varying global wind @ 36 fps.

flickers realistically on the reindeer and cobblestone floor.

In figure 7.26 we show the effects of using stationary and moving dynamic lights when visualizing the results of the 3D fire simulation affected by global wind. In the leftmost image we see that much of the illumination from the fire is lost because the three stationary dynamic point lights used are positioned along the central axis of the fire simulation, whereas the fire itself is affected by wind. This effect is avoided when using moving lights that follow the velocity field from the fire simulation, as can be seen in the center and rightmost images of the figure. However, when using few moving lights we get a repetitive pulsating effect as the lights are repositioned at the base of the fire when they move outside the fire. This pulsating effect is avoided when increasing the moving light count to 15, but this nearly cuts the frame rate in half.

### 7.2.7 Volume rendering

Here we compare the visual results from using respectively volume rendering and particle systems. As we do not visualize smoke using volume rendering, for comparison purposes we do not use a smoke particle system either. Figures 7.27 and 7.28 show two scenes comparing volume rendering and particle system visualization. For both the torch and campfire scenes the 3D fire simulation at 24x36x24 was used. As can be seen, volume rendering lacks the low-level detail caused by the textured particles in the particle systems. The volume rendering of the fire has a very smooth and transparent look. It might be possible to reduce the transparency by using different blending approaches, and the smoothness could be slightly reduced by using very large simulation grids in the underlying fire simulation. However, combining the volume rendering with a large simulation grid is not possible in real-time with current hardware.

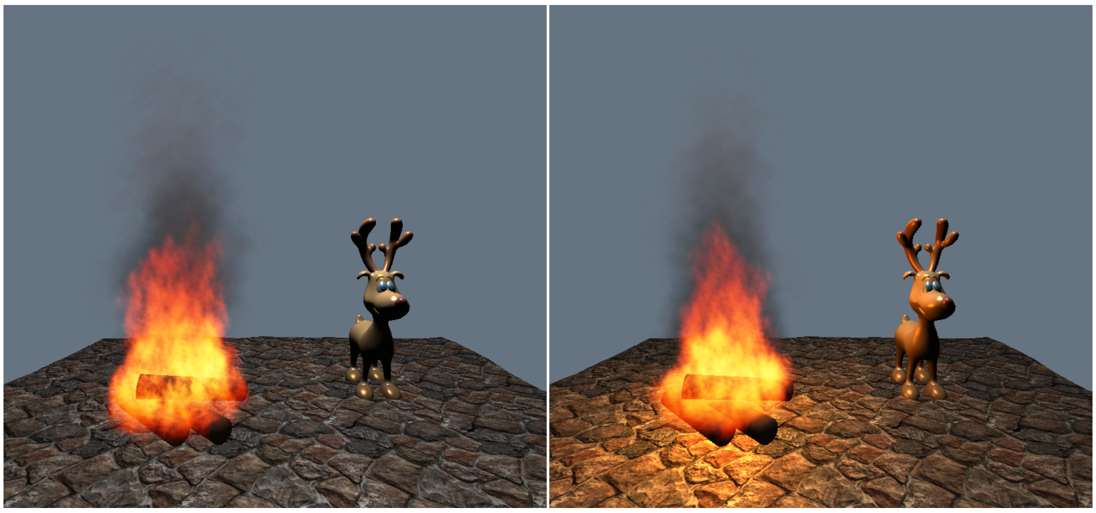

Figure 7.25: Dynamic illumination using the 2D fire slice simulation. Left: no dynamic lights, one static light @ 65 fps, right: three dynamic lights, one static light @ 47 fps.

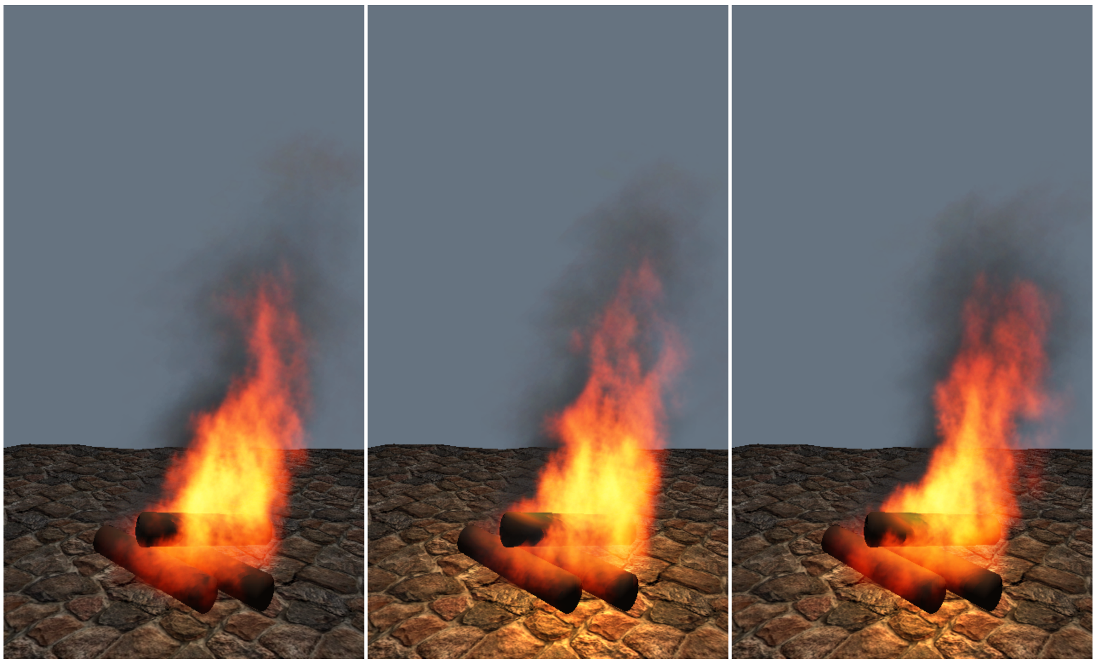

Figure 7.26: Dynamic illumination using the 3D fire simulation. Left: 3 stationary dynamic lights, one static light @ 42 fps, center: 3 moving dynamic lights, one static light @ 42 fps, right: 15 moving dynamic lights, one static light @ 22 fps.

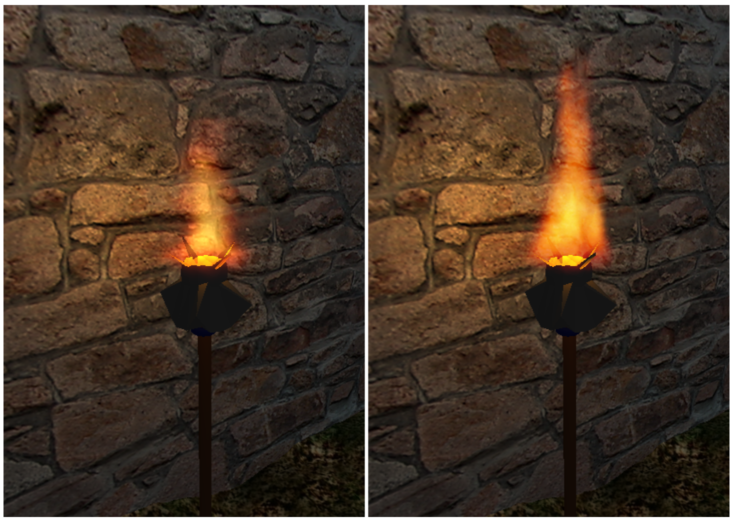

Figure 7.27: Torch visualization using the 3D fire simulation. Left: volume rendering @ 27 fps, right: fire particle system @ 36 fps.

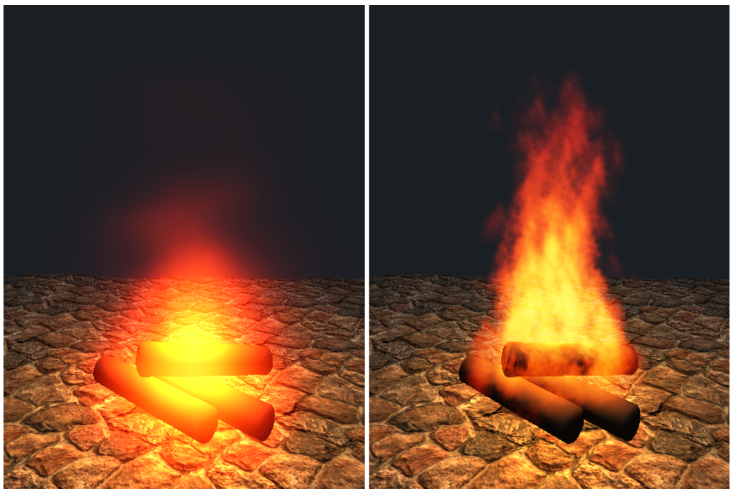

Figure 7.28: Campfire visualization using the 3D fire simulation. Left: volume rendering @ 31 fps, right: fire particle system @ 60 fps.

### 7.2.8 Vorticity confinement

The effect of the vorticity confinement force on the velocity field is very important to achieve turbulence and flickering in the fire. In figure 7.29 we show the effect on the 3D fire simulation of varying the $\epsilon$ parameter controlling the strength of the vorticity confinement, as presented earlier in equation 4.2.10. In the leftmost image we show the effect of disabling the vorticity confinement force by setting $\epsilon = 0$. As can be seen, neither turbulence nor flickering is present in the fire, thus making it look very unrealistic. In the rightmost image a very high vorticity confinement force is used, causing a very chaotic fire and reducing the effect of the crucial buoyancy force. In the center image we have used the same strength of the vorticity confinement force as for the rest of the visual results using the 3D fire simulation. This strength seems to be a good choice as the fire exhibits turbulent and flickering behavior without becoming too chaotic.

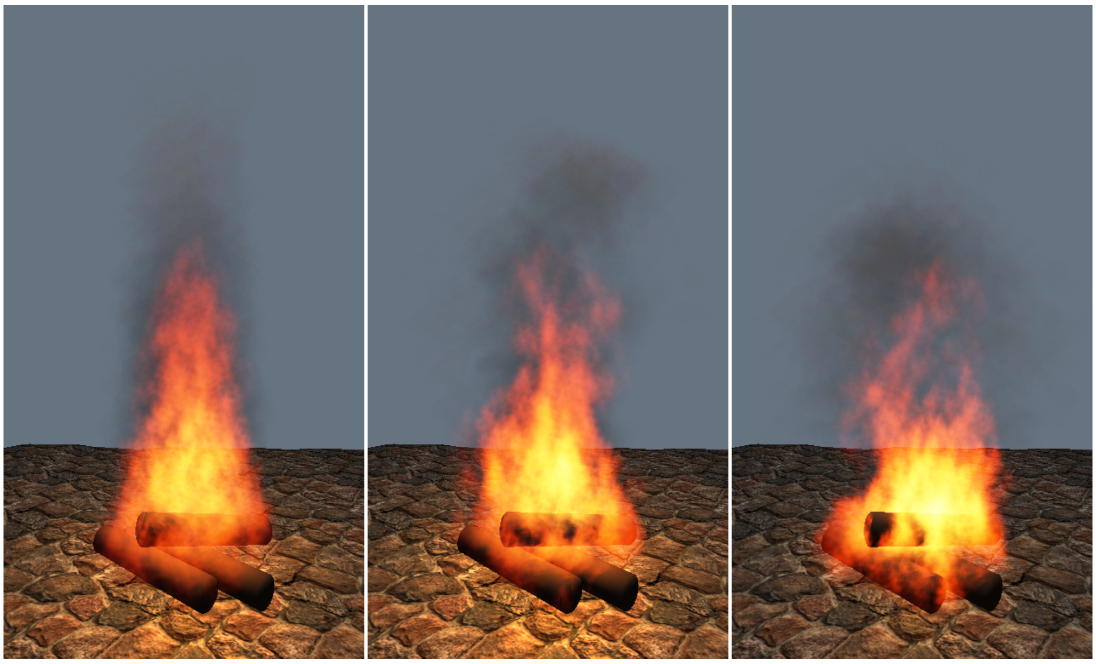

Figure 7.29: Various strengths of the vorticity confinement force using the 3D fire simulation. Left: $\epsilon = 0$ @ 42 fps, center: $\epsilon = 16$ @ 42 fps, right: $\epsilon = 40$ @ 42 fps.

### 7.2.9 Dynamic vs fixed timestep

In figure 7.30 we show the effect on the visual quality when using a dynamic timestep instead of a fixed timestep, which we have used in all visual results so far. We have reduced the window dimensions to 800x600 to get higher frame rates, as higher frame rates more clearly illustrate the differences between using a dynamic timestep and a fixed timestep. When a dynamic timestep is used the fire simulation is run each frame, making the simulation more accurate but also very different from the simulation using a fixed timestep of $\frac{1}{30}$ s. The figure clearly shows these differences. As we have adapted the simulation parameters for a simulation running at 30 FPS, using a dynamic timestep is not very convenient when the frame rate differs significantly from 30 FPS. When running the simulation and visualization on different hardware, using a dynamic timestep can also lead to unpredictable behavior due to differing frame rates. However, as we have not implemented interpolation between simulation frames, a fixed timestep of $\frac{1}{30}$ s causes the animation to be a bit choppy due to the asynchronous simulation and visualization.

In the rightmost image we show the effect of reducing the timestep to $\frac{1}{100}$ s running the simulation at 100 FPS. As the simulation gets more accurate we have to increase the buoyancy force to get a similar fire as the one in the center image. Even though the decrease in frame rate is noticeable, the gain in visual quality compensates for the reduced frame rate. Especially the high-level turbulence and flickering is better reproduced in rightmost fire. As the processing time used for the 2D fire slice simulation is rather low compared to the 3D fire simulation, a lower fixed timestep is possible while stile achieving real-time frame rates. A lower timestep also reduces the tendency of animation chopping as experienced when using a timestep of $\frac{1}{30}$ s.

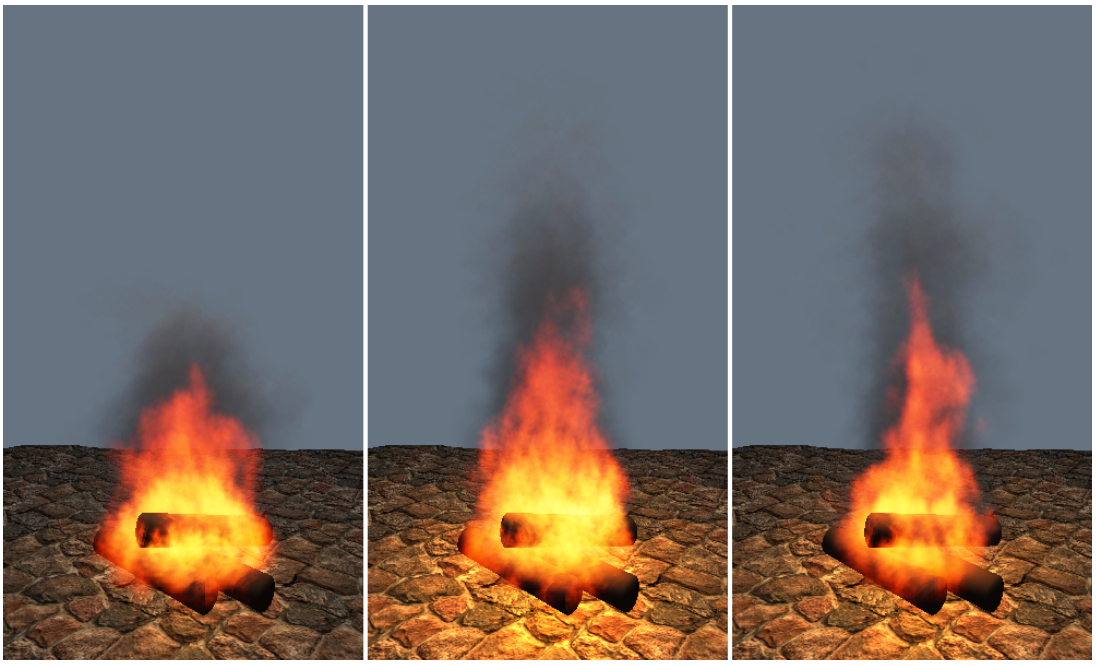
Figure 7.30: Dynamic vs fixed timestep using the 2D fire slice simulation and window dimensions of 800x600. Left: dynamic timestep @ 60 fps, center: fixed timestep of $\frac{1}{30}$ s @ 66 fps, right: fixed timestep of $\frac{1}{100}$ s @ 53 fps.

### 7.2.10 Fire color scale

Here we show the results of varying the fire color scale $T_{scale}$, controlling the overall brightness of the fire. $T_{scale}$ is used in equation 5.2.3 when computing the fire color field. Figure 7.31 shows the fire visualized using the 3D fire simulation at three different fire color scales, increasing from left to right. As the fire might be used in a lot of different environments, it is important to be able to control its brightness. A real fire will for example look different at day and night because the human eye adapts to the overall illumination in the environment. Thus, a bright flame might look more realistic in a dark environment, and vice versa for bright environments. As the fire color scale can be interactively adjusted, it could be used to dynamically update the brightness of the fire based on the environment.

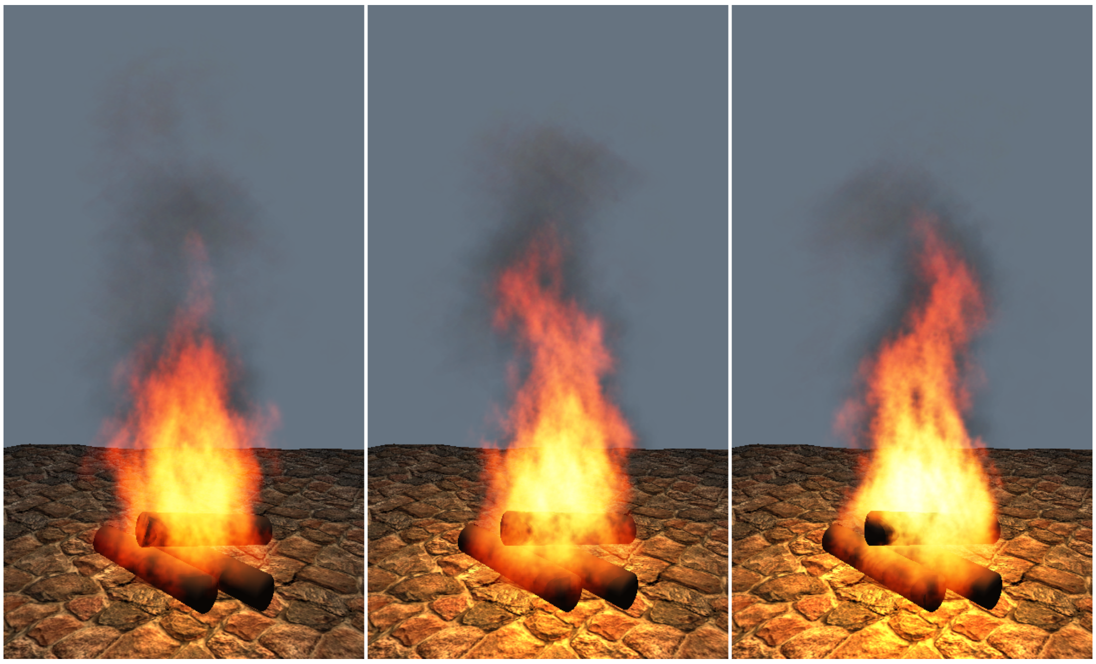

Figure 7.31: Various fire color scales using the 3D fire simulation. Left: $T_{scale} = 0.065$ @ 42 fps, center: $T_{scale} = 0.08$ @ 42 fps, right: $T_{scale} = 0.095$ @ 42 fps.

## 7.3 Comparison with footage from real fire

In this section we compare our fire to footage from real fire, looking at strengths and weaknesses of our fire visualization. We use images of real fire to guide the comparison, in addition to video footage[2] of the real fire in figures 7.32 and 7.33. For each of the three real images we have simulated and visualized our fire using a 3D simulation grid at 24x36x24, a fire particle system of 2048 particles, and a smoke particle system of

[2]The video footage of a real fire can be seen in the accompanying *media* folder.

512 particles. In order to achieve an appearance similar to the real fire we have also adjusted our fire simulation parameters for each of the images. We present the three real images along with screen captures from our fire in figures 7.32, 7.33, and 7.34 [3]. It should be noted that still images do not do full justice to our fire, as it looks a lot better in motion.

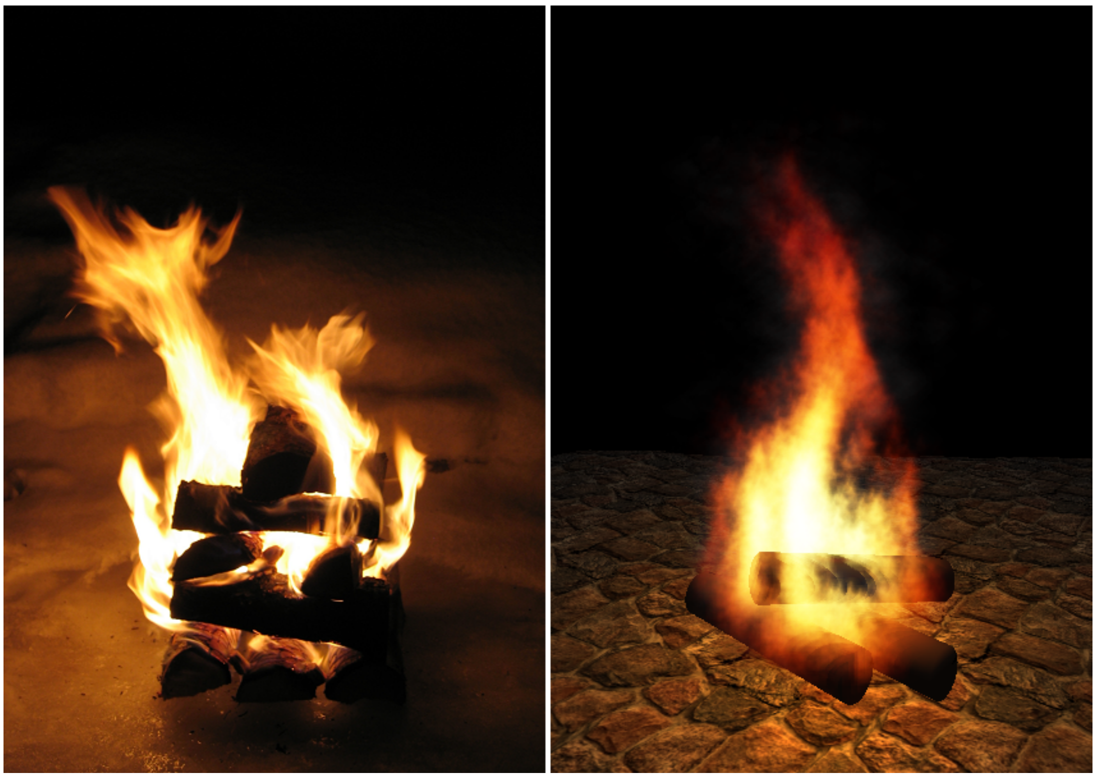
Figure 7.32: Left: real footage of a turbulent fire, right: particle system visualization of our fire @ 40 fps.

### 7.3.1 Color and brightness

As can be seen from the images of real fire, the perceived fire color and brightness are very dependent on the overall illumination in the environment where the fire is located. The real fire in figure 7.32 has a very high perceived intensity, and we have tried to create a similar fire by increasing $T_{scale}$ used when computing the fire color field. As can be seen, the color and brightness near the center of our fire is realistically reproduced, whereas the color at the edges of the flames is a bit too red.

In figures 7.33 and 7.34, the fires are located in environments with more overall illumination, and the effect is fire colors that are less intense and have more color variations.

[3]The photography of the campfire was retrieved from
`http://commons.wikimedia.org/w/index.php?title=Image:Campfire_4213.jpg&oldid=811870`

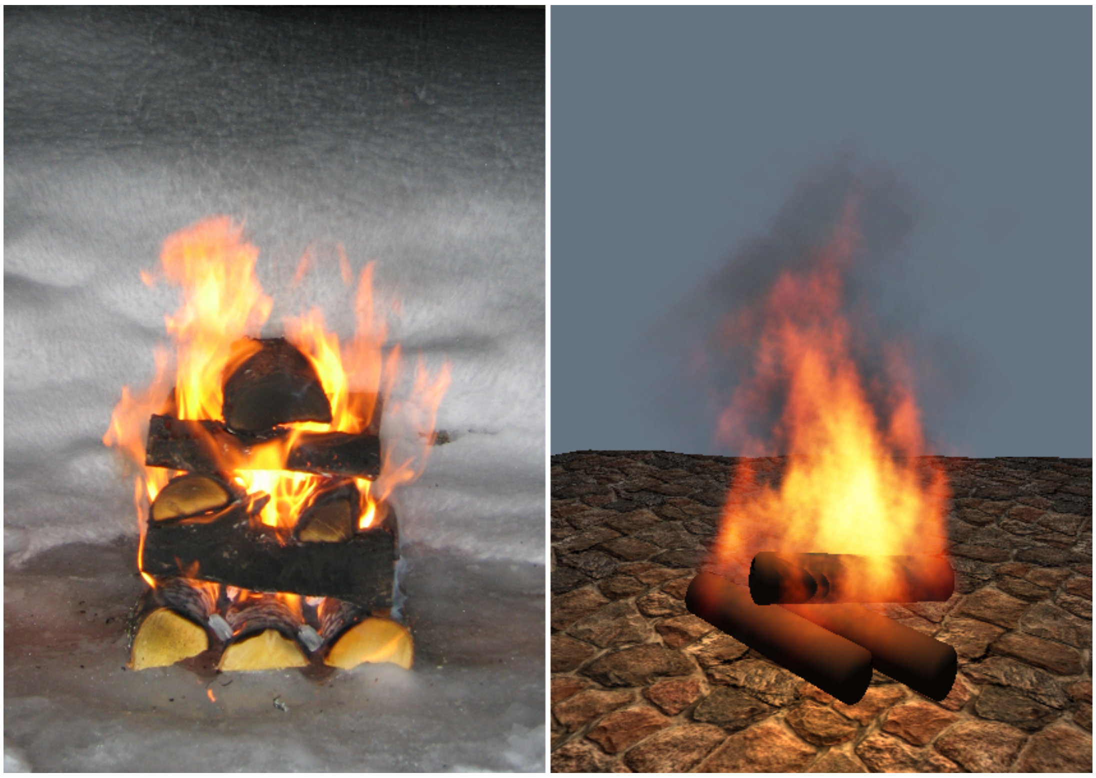
Figure 7.33: Left: real footage of a turbulent fire using flash to light the scene, right: particle system visualization of our fire @ 40 fps.

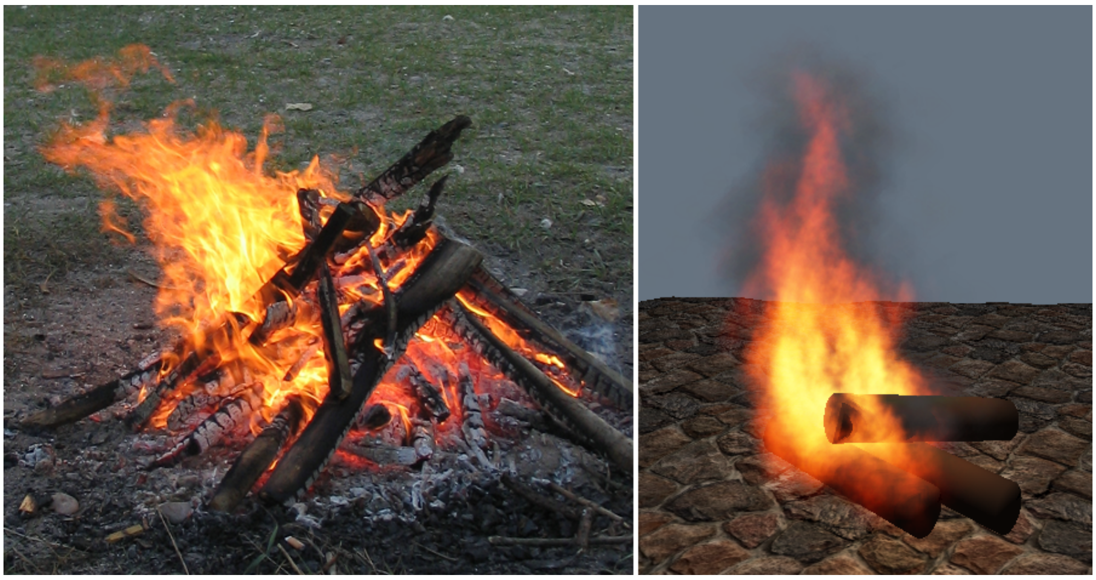
Figure 7.34: Left: real footage of a campfire, right: particle system visualization of our fire @ 40 fps.

It is clear that our fire has more realistic colors in bright environments than in dark environment. The color at the edges of the flames are still a bit red, but not as much as in figure 7.32.

### 7.3.2 Texture and low-level detail

The real fire in figure 7.32 has both texture and low-level detail as evident by the color variations in the flame surfaces and the irregularities of the flame shapes. Still, the flames have a smooth appearance and sharp edges. Similar properties can be seen in the real fire in figure 7.33, whereas the real fire in figure 7.34 can be said to have even more texture and low-level detail.

Our fire fails to capture some of the low-level detail and texture of real fire. The surfaces of our flames are rougher than the smooth surfaces of real fire, due to the use of textured particles in the particle system visualization. Textured particles also cause our flame edges to look more fuzzy, and we fail to capture the sharp edges of real flames. Finally, the color variations in our fire are also more diffuse. Even though our fire does not look completely like real fire, the use of textured particles does lead to a certain amount of texture and low-level detail.

### 7.3.3 Motion and flame shape

Based on video footage of real fire we have observed the following properties of the motion and shape of fire:

- Large variations in flame shape within small time frames.
- Very rapid flickering and turbulence combined with buoyant upward motion.
- Small flame plumes detach from the main flame body before fading.
- The flame is greatly influenced by gusts of wind, and can sway rapidly from side to side.
- Fire consists of many small flames which can move in individual directions.

In comparison, our fire exhibits both flickering and turbulent features as well as swaying and buoyant upward motion. Real fires are however more chaotic than what we have been able to model, especially with regard to the large flame shape variations within small time frames. Also, our fire only to a small degree exhibits the separation of small flame plumes from the flame body. By using a dynamic wind boundary we can model wind gusts, but we have not been able to get the fire to sway from side to side as rapidly as real fires. The flames also move more uniformly than in a real fire. Still, the flickering motion and shape of our fire is partially reminiscent of real fire.

### 7.3.4 Dynamic illumination

As seen in figure 7.32 and also earlier in figure 7.25, our dynamic illumination does a good job of approximating the illumination caused by real fire. The color and brightness of the dynamic illumination closely match that in the real footage. Based on the video footage of real fire we have also observed that the flickering of illumination on the surrounding scene due to the variations of the fire is also quite realistic in our visualization. Stationary lights are however a bit lacking when the fire sways from side to side. A larger number of moving lights do a better job of representing the illumination by the fire, albeit at a lower frame rate.

## 7.4 Comparison with other approaches

In this section we briefly compare our visual and performance results against other previously mentioned physically based approaches for simulating and visualizing fire. To guide the comparison we have used still images [4] and video captures [5] from the other approaches. As for the comparison with real fire we have defined a set of criteria to guide the visual comparison: flickering, turbulent behavior, color, texture, and smoke.

Flickering and turbulent behavior as well as a smooth appearance with realistic colors is best achieved by the offline method presented in [NFJ02], producing visual results that are close to that of real fire. As their fire effects are not simulated nor rendered in real-time a direct comparison would not be fair.

Our visual results compare favorably to the results in [MK02], [WLMK02], [KW05], and [IMDN05]. The focus in [MK02] is on fire simulation and not visualization, and their visual results are therefore not convincing. Like our fire visualization, the one in [WLMK02] has some low level texture, but the colors are more bland than ours even though they also use a black-body radiation table to compute the fire colors. When it comes to motion, their fire fails to capture the crucial flickering and turbulent elements found in real fires. We have been able to reproduce these elements quite a bit more realistically. The motion of their fire is also slightly periodic. Of the four approaches mentioned they have a smoke visualization closest to ours.

The fire in [KW05] also fails in capturing the flickering and turbulent elements as

[4]We presented still images from each approach in chapter 3.

[5]The video captures used can be found at the following locations (as of 12-Jun-2006):

- [NFJ02]: `http://graphics.stanford.edu/~fedkiw/animations/campfire.avi`
- [MK02]: `http://people.cs.tamu.edu/z0m8905/papers/tvcg.mov`
- [WLMK02]: `http://www.cs.sunysb.edu/~vislab/projects/amorphous/WXMWebsite/fireimg/torchnnnn3.avi`
- [KW05]: `http://wwwcg.in.tum.de/Research/data/Publications/eg05/low.avi`

We have not found any video capture for the approach presented in [IMDN05].

convincingly as ours. The buoyant upward motion is also too slow. In addition, their fire has quite saturated colors and does not look very natural. The fire colors in [IMDN05] are also a bit saturated, but in contrast to the other three approaches they include dynamic illumination of the surrounding scene. The effect of the dynamic illumination is more realistic than the fire itself.

Our performance results are also competitive with previous approaches, even when taking into account the variations in hardware. [MK02] achieved 20 FPS with a 3D fire simulation at 20x20x20 on the CPU. With the same grid dimensions our simulation runs at around 280 FPS, which shows that large gains in performance can be achieved by utilizing the GPU. Using the LBM to perform the fluid simulation, [WLMK02] achieved a frame rate of around 215 FPS for 3D fire simulation at 32x32x32 on a NVIDIA GeForce3 Ti200. Our fire simulation runs at 90 FPS with the same grid dimensions on more powerful hardware, which shows that the LBM is a lot faster than the stable fluid solver, although the accuracy and flexibility of the former is unknown. When rendering 100 texture splats on top of the fire simulation, the frame rates of [WLMK02] drop to 65 FPS. Again using the same simulation, we are able to render 1024 fire particles at 65 FPS using window dimensions of 1024x768. Although our approach would perform a lot worse than theirs on similar hardware, this is justified as our visual results are better.

Next, [KW05] achieve a simulation frame rate of 190 FPS on a NVIDIA 6800 GT when performing a 2D slice simulation with two slices at 64x64 each. With the same slice count and grid size our simulation runs at about 425 FPS. However, we do not compute pressure templates, nor do we explicitly extrude the 3D volume. As they have not mentioned their frame rates when both simulating and visualizing fire, how their fire visualization performs compared to ours is unknown.

Finally, [IMDN05] achieve a processing time of 1.3 s when using a simulation grid size of 32x32x32 and illuminating a room by using point lights positioned at voxel centers. Thus, their approach is clearly not real-time like ours.

## 7.5 Summary

In this chapter we have first presented performance and visual results from the simulation and visualization of fire. The performance results show that the visualization of the fire tends to be the bottleneck at sufficiently large simulation grids, especially when using the 2D fire slice simulation, which is more efficient than the 3D fire simulation. Increasing the particle counts used when visualizing the fire does not cause a linear increase in rendering time, as the particle sizes must be decreased to avoid saturation. Next, we have found that the frame rates are highly dependent on the the window dimensions and number of point lights used, due to the higher processing time needed for visualizing the fire and the surrounding scene. The processing time required for the fire simulation is however independent of the window dimensions. We have also noted that a fixed timestep increases the frame rate at high frame rates and reduces the frame rate at low frame rates with respect to a dynamic timestep, as the simulation is run a varying of number of times per frame depending on the timestep. Finally, the results clearly show that our volume rendering approach is not suitable for real-time visualization on current hardware when the fire covers more than a small part of the viewport.

We have also made several important observations from the visual results:

- Medium sized simulation grids produce the best trade-off between low-level and high-level turbulence.
- 2D fire slice simulation only works well when rotationally symmetric fire effects are visualized, but in this case only one or two slices are necessary for good visual results.
- The 3D fire simulation is best suited for simulating different kind of fire effects like global wind and simulation domain advection.
- Simulation domain advection makes the fire trail behind realistically as it is moved.
- Smoke enhances the fire visualization in bright environments by acting as a backdrop for the fire.
- Dynamic illumination greatly increases the immersion of an environment.
- Our volume rendering approach is not capable of visualizing fire with good visual quality.
- Vorticity confinement is crucial to achieve the necessary amount of turbulence in the fire.
- A fixed timestep makes the fire behave predictably, whereas a dynamic timestep gives different results depending on the frame rate.
- The fire color scale can be used to adapt the fire for dark or bright environments.

Finally, we have compared our fire to footage of real fire. We have noted that although our fire can not be confused with real fire, it looks a lot better in motion than in still pictures. While not at the level of real fires, our fire exhibits realistic colors and enough turbulence and flickering to produce visually pleasing results. The dynamic illumination is especially reminiscent of illumination from real fire. Our fire also compares favorably to previous approaches for simulating and visualizing fire.

# Chapter 8

# Discussion

In this chapter we first generally discuss the performance and visual results presented in the previous chapter. Next, we evaluate the results based on the requirements presented in section 1.2. We then present our main contributions, and finally we conclude the thesis and present ideas for future work.

## 8.1 General

We have compared the volume rendering and particle system visualization methods, and based on our visual results we have concluded that particle systems are far more suitable for fire visualization than volume rendering. As we achieved both low frame rates and low visual quality when volume rendering the fire, we chose not to focus any effort into volume rendering smoke. Still, volume rendering might have potential for certain smooth and small-scale fire effects like candles.

Next, we chose to use separate particle systems for visualizing the fire and the smoke. Using the same particle system by incorporating the smoke directly into the fire color field would make it impossible to balance the particle counts and particle sizes for fire and smoke individually. This balancing option is important, as we have found that the smoke works better with larger particle sizes and lower particle counts than the fire. Using a single particle system would also have made it hard to find good blending approaches suitable for both the fire and the smoke. By rendering the smoke before the fire and using appropriate blending approaches, we avoid having to sort the particles based on their distance from the eye position. As the emissive fire is rendered on top of the absorbing smoke, brighter backgrounds can be used than would be possible when only rendering fire.

We have also compared two main simulation approaches: 3D fire simulation and 2D fire slice simulation with volumetric extrusion. We have found that the 3D fire simulation both produces good visual results and is capable of simulating a wider range of effects like moving torches and fire affected by dynamic wind. However, the 2D fire slice simulation is a bit more efficient than the 3D fire simulation, and works quite well when simulating rotationally symmetric phenomena. Thus, if the fire simulation should become the bottleneck, for example if the fire will only be viewed from a distance, the 2D fire slice simulation can be preferable.

From experimenting with different simulation grid sizes, we have come to the conclusion that the scale of the turbulence in the fire simulation is highly dependent on the simulation grid size. Thus, larger simulation grid sizes do not necessarily produce fires with more realistic motion. Larger simulation grids produce more low-level turbulence instead of the high-level turbulence important for achieving a flickering and swaying fire. Using different closed boundary conditions does not seem to affect the turbulence of the fire, likely because the boundary conditions only affect the turbulence at the boundary of the computational domain, whereas the fire is located in the center. However, the dynamic wind boundary can to a certain degree increase the swaying of the fire when using a 3D fire simulation, thus creating the illusion of more turbulence. Our experiments with mass expansion have shown that it works as expected, by causing the fire to expand in the area where combustion occurs. However, we have found that it is usually not needed, as we do not have problems with the fire becoming too thin. The vorticity confinement scaling shown in equation 4.2.11 has been used successfully to increase the large-scale turbulence of pure 2D fire simulations, but it does not seem to make much of a difference in the 2D slice or 3D case.

The simulation parameters that are most crucial for achieving realistic fire motion and turbulence are buoyancy and vorticity confinement. We have found that the gravity parameter is rather insignificant in this context, which is logical as the buoyancy is the dominating force in a real fire. The dissipation and diffusion parameters must be used in combination with the fuel gas sources in order to adjust the extent of the exhaust gas and heat, thus controlling the height and range of the fire. These parameters can also be used to control the height of the smoke above the fire, albeit to a rather low degree.

Dynamic illumination has the potential to greatly increase the immersion of an environment, as it looks unnatural if the fire does not affect the illumination of the rest of the scene. We have found that even just a few stationary dynamic point lights positioned in the center of the fire can produce convincing illumination. When the fire sways due to movement or wind, moving point lights would be preferable, but in order to avoid pulsating effects a high amount of moving point lights are needed, which drastically reduces the frame rate. Thus, it is usually better to use a set of strategically positioned stationary dynamic point lights.

By comparing fixed and dynamic timesteps we have come to prefer using a fixed timestep, as this causes the simulation to be predictable regardless of the application frame rate. This predictability can be crucial, especially in situations where the application might be run on a wide range of different hardware. Even on the same hardware, variations in the frame rate during the simulation could cause the fire to behave unpredictably. With a fixed timestep and equal starting conditions, two simulations would produce the same results even if run for a long time, which would not be possible with a dynamic timestep. The fixed timestep can be adjusted to balance between simulation efficiency and accuracy. We have found that we can achieve better turbulence and motion in the 2D fire slice simulation by lowering the fixed timestep used, thus increasing the visual quality. As the 2D fire slice simulation is very efficient, we achieve real-time frame rates even at low timesteps, as shown earlier in figure 7.30. We have not experienced a corresponding increase in visual quality when lowering the fixed timestep used for 3D fire simulation, which also causes the frame rates to drop quite fast.

## 8.2 Evaluation

In this section we evaluate our performance and visual results based on the requirements presented in section 1.2.

**R1: A physically based fire simulation should guide the visualization.**

In order to model the physics of fire we have developed a physically based simulation algorithm, as presented in chapter 4. Heat transfer and mass transfer are modeled by a fluid simulation in combination with buoyancy and vorticity confinement controlling the amount of upward motion and turbulence in the fire. The fluid simulation models the air in which hot exhaust and fuel gases flow, whereas combustion and fuel sources control the amount of fuel and exhaust gases as well as the temperature of the fire. We do not explicitly simulate the reaction zone and thus the blue core of the fire, but instead use an implicit combustion model which consumes fuel gas when the temperature exceeds the fuel ignition temperature, creating exhaust gas and more heat. We do not model the burning of objects or consumption of fuel sources, as we use a static fuel source field which is constantly injected into the fuel gas field. Even though the physical correctness of our fire simulation could be increased, it is still physically based and fulfills the requirement.

**R2: The fire simulation and visualization combined should be able to run in real-time.**

For our purposes we defined real-time to be 30 frames per second. As the results presented earlier in section 7.1 confirm, we have reached and exceeded 30 frames per second for simulation and visualization of fire combined, even when dynamically illuminating a surrounding environment. Our approach is therefore perfectly capable of running in real-time on modern graphics hardware without claiming all the computational resources, thus fulfilling the requirement.

**R3: The fire should be visually convincing and must therefore have a realistic appearance, close to that of real fire.**

We listed three demands to guide the evaluation of this requirement:

- The intensity (color and brightness) of the fire should be realistic.
- The fire should have realistic texture and low-level detail.
- The overall shape of the fire should be believable.

By using a physically based black-body radiation model to compute the fire colors of the fire, we feel that we have achieved a rather good approximation of the color and brightness of real fire. Texture and low-level detail has been achieved using textured particles in the fire particle system. The underlying fluid simulation used to simulate the motion of the fire allows us to create believable fire shapes by adjusting the simulation parameters.

Although we have achieved good visual results suitable for inclusion in games or virtual environments, it is not hard to tell the difference between our fire and a real fire, as shown in section 7.3. Still, our fire definitely compares favorably with previous work in real-time fire simulation and visualization, and we have partially fulfilled the requirement.

**R4: The fire should move and behave realistically. This requires capturing the flickering and turbulent characteristics of fire.**

We have attempted to capture the motion of fire by using a fluid simulation to model the mass and heat transfer properties of fire. Buoyancy makes the flames rise due to heat, and vorticity confinement increases the turbulence of the fire. Our fire simulation is thus capable of modeling the flickering and turbulent characteristics of fire, although not to the same degree as experienced in real fires, as we mentioned earlier in section 7.3. Still, the motion is convincing enough to be able to compete favorably with previous fire effects used in games and virtual environments, thus partially fulfilling the requirement. We also find it important to repeat that the fire looks better in motion than in still captures.

**R5: Smoke should be incorporated in the visualization.**

We listed three demands to guide the evaluation of this requirement:

- The smoke should have realistic appearance, including intensity, texture, and shape.

- The smoke should have believable motion.
- The coupling between fire and smoke should be realistic.

We have incorporated light-absorbing smoke in the visualization. The smoke particles are rendered before the fire particles, causing the fire and smoke to blend realistically. The semi-transparency and fuzzy edges of the visualized smoke can be compared to real smoke. The intensity and shape of the smoke are also visually convincing, and the smoke moves realistically due to the fluid simulation.

Because the fire and smoke use the same underlying simulation and are both computed based on the exhaust gas and temperature fields, the coupling between fire and smoke works good. A drawback is that as smoke is rendered before fire particles, we can not model more explosive and large scale fire effects where the smoke is heavier and more visible throughout the fire. Also, it is hard to control the height and range of the smoke without affecting the fire, thus the smoke tends to fade out a bit too soon. All in all, we have fulfilled this requirement to a partial degree.

**R6: The user should be able to interact with the fire in some fashion.**

We do not model physical interaction between the fire and objects or scene geometry. It is however possible to interact with the fire to a limited degree, as the fire reacts realistically as it is moved, as shown in section 7.2.4. Thus, we have fulfilled this requirement to a partial degree.

**R7: It should be possible to use the fire in a virtual environment.**

This requirement listed two demands:

- The fire should be realistically incorporated in the surrounding scene.
- There should be computational power left for visualizing the virtual environment. The fire should not claim all the computational resources.

As shown in section 7.2, we have realistically incorporated the fire into several scenes. In addition to simply including the fire in the scene, we also model the illumination caused by the fire on the surrounding environment. The dynamic illumination causes the coupling between the fire and the scene to feel more natural. We achieve real-time frame rates even when dynamically illuminating the surrounding environments, and the fire is therefore suitable for use in most virtual environments without claiming all the computational resources, thus fulfilling the requirement.

## 8.3 Contributions

In the course of this thesis we have combined ideas and methods from previous approaches in new ways, modified previous approaches to run on the GPU, and come up with our own extensions to previous work. Our main contributions to simulation and visualization of fire in real-time can be summarized as follows:

- We explicitly perform the combustion process and fluid simulation from [MK02] on the GPU[1].
- We visualize the fire using separate particle systems for fire and smoke. Previously, a single particle system has been used for both fire and smoke visualization by among other [WLMK02] and [KW05].
- We have compared particle systems and volume rendering, and have established that particle systems are currently better suited for real-time visualization of fire.
- We use vorticity confinement from [FSJ01] in combination with the combustion process and fluid simulation from [MK02] to increase the turbulence of the fire.
- We use an approximation of the approach in [IMDN05] to incorporate real-time dynamic illumination in the visualization by strategically positioning a set of stationary dynamic point lights in the fire simulation domain, modeling the flickering illumination on other objects in the scene caused by the fire. We have also implemented moving dynamic point lights which follow the velocity field of the fire simulation, thus giving a more accurate representation of the fire illumination.
- We use the volumetric extrusion from [RNGF03] in combination with the combustion process and fluid simulation to increase performance for fire effects with rotational symmetry.
- We have implemented a novel method which we call simulation domain advection, used to move the simulation domain in order to follow the movement of the fire, thus modeling effects such as trailing flames as a torch is moved. This method is an improvement of the method used by [WLMK02] to achieve the same effect. Unlike our approach which is more physically correct, they set a static wind boundary condition proportional to the speed at which the fire is moved.
- We use the wind boundary condition adapted from [AL04] with a wind vector based on time-dependent noise curves to model dynamic wind affecting the fire.

We have submitted a paper to the conference *Theory and Practice of Computer Graphics 2006*[2] and have gotten it accepted for presentation and publication. In the paper

[1][MK02] performed the combustion process and fluid simulation on the CPU.
[2]`http://www.eguk.org.uk/TPCG06/`

we presented the physically based simulation and the particle system fire visualization and performed a comparison between the 3D fire simulation and the 2D fire slice simulation. The paper is appended at the end of this thesis.

## 8.4 Conclusion

We have developed and implemented a physically based approach for simulating and visualizing 3D fire in real-time. By using vorticity confinement in combination with the combustion process and fluid simulation from [MK02], we have achieved a turbulent underlying fire simulation which can be performed both in 2D and 3D. Volumetric extrusion allows us to implicitly perform a 3D fire simulation by simulating a set of independent 2D slices, which is a lot more efficient in terms of computational requirements. The results from the fire simulation are then visualized using two particle systems, one for fire and one for smoke. The particle systems sample the particle colors from the fire color and smoke fields, which are computed based on the exhaust gas and temperature fields from the fire simulation, and the particles are moved using the velocity field from the fire simulation. We have also implemented a volume rendering approach, but it falls short both in performance and visual quality. The main fire simulation and visualization has been extended with dynamic illumination, simulation domain advection, and dynamic wind, to better model the interaction between the fire and the user and surrounding environment.

By taking the laws of nature and fire physics into account when performing the physically based simulation, we have arrived at a flexible and extensible solution for simulating realistic small-scale fires. To be able to achieve real-time frame rates, we have implemented both the simulation and visualization of fire completely on the GPU by utilizing its programmability and computational power.

We have either fulfilled or partially fulfilled all of the requirements we listed at the beginning of the thesis. Although our fire can not be confused with real fire, it is visually appealing both in appearance and motion, and compares favorably to other real-time approaches. Our approach is also more flexible than previous approaches like animated textures and particle systems with no physical basis. Compared to previous physically based real-time approaches, our fire has more realistic motion with higher amounts of turbulence and flickering. The fire is definitely suitable for inclusion in virtual environments and games and should be capable of increasing the immersion of the environments, especially in combination with dynamic illumination.

As GPUs become increasingly powerful, both fire effects and visualizations of other natural phenomena should become more and more realistic while still running in real-time. These improvements will benefit computer graphics practitioners and researchers, video game enthusiasts, and other groups who in some way depend on interactive and realistic visualizations of natural phenomena.

## 8.5 Future work

There are many possibilities for improvements to the fire simulation and visualization approach we have presented in this thesis. With a focus on increasing the realism of the fire and its interaction with the environment, our ideas and suggestions for future work are as follows:

- The fire simulation is currently uncoupled from objects that are supposed to be burning. An idea could be to make it look more like the actual objects are burning by emitting particles from the polygons of the objects. The particles could initially be weighted to follow the normal of the polygon, before gradually following the velocity field of the fire simulation. In addition, the fuel source field could be automatically initialized based on the polygon meshes by computing which grid cells are contained in the polygon mesh.
- Another extension coupled to the burning of objects would be to model the deformation of the polygon meshes of the objects that are supposed to be burning, as done by [MK05].
- By modeling arbitrary boundaries in the fluid simulation, as done by [WLL04], the interaction between the fire and objects could be made more realistic. Arbitrary boundaries would cause the flames to move naturally around obstacles and could potentially increase the turbulence and flickering of the fire.
- We do not currently have much separate control of the smoke, as it is based on the exhaust gas and temperature fields, just like the fire. An idea could be to model the smoke with a separate field, making it easier to adjust both the height and range of the smoke and the amount of smoke produced.
- Instead of performing the exponential mapping when calculating the black-body radiation table, a more physically correct visualization could be used by storing the original black-body intensities in the lookup table, and instead using a high dynamic range (HDR) when rendering the fire particle system. To avoid clamping the particle colors, the particle system could be rendered to a high-precision off-screen texture and finally exponential mapping could be used before blending the offscreen texture with the framebuffer.
- As mentioned earlier we have to make a trade-off between high-level and low-level turbulence. Instead, it might be possible to use an additional velocity field template to model the low-level turbulence while modeling high-level turbulence with a rather small simulation grid. The velocity field template should be able to be tiled periodically in space and could be computed using a three-dimensional Kolmogorov spectrum as done by [SF93]. Thus, when sampling fire particle velocities the contributions from the velocity field template and the velocity field from the fire simulation are added together, achieving both low-level and high-level turbulence.

- As we suggested earlier, the particle sizes and particle counts of both the fire and smoke particle systems could be adjusted using a level-of-detail approach based on the distance between the view position and the fire, thus reducing fill-rate and vertex processing requirements when the fire is further away.
- An extension of the visualization could be to model the heat shimmer caused by the hot air rising above the fire, as done by [Ngu04]. Heat shimmer in combination with dynamic illumination are the two most important visual side effects which due to the fire affect the surrounding scene.
- By using a different attenuation function than quadratic fall-off to model the dynamic illumination of the point lights used to light the scene, it would be possible to create a bounding box surrounding the fire outside of which there would be no illumination contributions from the fire. This bounding box could be used to avoid having to compute dynamic illumination for the entire scene if only a small part would be affected by the fire.
- The dynamic illumination could be improved by incorporating shadows caused by blocking objects in the scene.

# Bibliography

[AGE06] Ageia, 2006. [Online; accessed 15-May-2006]. `http://www.ageia.com/`.

[AL04] Michael Aagaard and Dennis Lerche. Realistic modelling of falling and accumulating snow. Master's thesis, Laboratory of Computer Vision and Media Technology, Aalborg University, 2004.

[Alm05] Espen Almdahl. Videreutvikling av sverresborgmodellen. Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2005.

[Bat00] G.K. Batchelor. *An introduction to fluid dynamics.* Cambridge University Press, Cambridge, 2000.

[BD98] Roland Borghi and Michel Destriau. *Combustion and flames : chemical and physical principles.* Éditions Technip, 1998.

[BLLR06] Flavien Bridault-Louchez, Michel Leblond, and Francois Rousselle. Enhanced illumination of reconstructed dynamic environments using a real-time flame model. In *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 31–40, New York, NY, USA, 2006. ACM Press.

[BPP01] Philippe Beaudoin, Sébastien Paquet, and Pierre Poulin. Realistic and controllable fire simulation. In B. Watson and J. W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 159–166, 2001.

[Cg06] Nvidia cg, 2006. [Online; accessed 08-Jun-2006]. `http://developer.nvidia.com/page/cg_main.html`.

[Com03] Houghton Mifflin Company. The american heritage® dictionary of the english language, fourth edition, 2003. [Online; accessed 15-May-2006]. `http://www.thefreedictionary.com/fire`.

[Com06] Introduction to computational fluid dynamics, 2006. [Online; accessed 15-May-2006]. `http://www.cham.co.uk/website/new/cfdintro.htm`.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[DHR$^+$02] Patrick Dalton, Shyh-Chyuan Huang, Rob Rosenblum, Lawrence Lee, and Hank Driskill. Digital pyro for "reign of fire". In *SIGGRAPH 2002 Visual Proceedings (sketch)*, page 167, 2002.

[Fal05] Bård Terje Fallan. Stedfestet lyd i modeller for virtuell virkelighet. Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2005.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 3–4. Addison-Wesley Professional, 2004. Part I: Natural Effects.

[Fie06] Glenn Fiedler. Game physics: Fix your timestep!, 2006. [Online; accessed 22-May-2006]. `http://www.gaffer.org/articles/Timestep.html`.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*, chapter 5: Lighting, pages 101–142. Addison-Wesley, 2003.

[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. *ACM Trans. Graph.*, 22(3):708–715, 2003.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press.

[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.

[Har05] Mark J. Harris. Mapping computational concepts to gpus. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 493–508. Addison-Wesley Professional, 2005.

[HB04] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL (3rd Edition)*, chapter 10: Illumination Models and Surface-Rendering Methods, pages 556–667. Pearson Prentice Hall, 2004.

[HBSL03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Hol03] Thorsten Holtkämper. Real-time gaseous phenomena: a phenomenological approach to interactive smoke and steam. In *Afrigraph*, pages 25–30, 2003.

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

[KCR99] Scott A. King, Roger A. Crawfis, and Wayland Reid. Fast animation of amorphous and gaseous phenomena. In *Volume Graphics '99*, pages 333–346, 1999.

[KF05] Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 471–491. Addison-Wesley Professional, 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. Massively parallel particle systems on the gpu. In Wolfgang Engel, editor, *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, pages 119–133. Charles River Media, 2004.

[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.

[LL98] L. D. Landau and E. M. Lifshitz. *Fluid Mechanics, 2nd edition.* Butterworth-Heinemann, Oxford, 1998.

[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. `http://www.cg.tuwien.ac.at/research/publications/1997/Matkovic-thesis/`.

[MK02] Zeki Melek and John Keyser. Interactive simulation of fire. *Technical Report, Computer Science Department, Texas A & M University*, pages 2002–7–1, 2002.

[MK05] Zeki Melek and John Keyser. An interactive simulation framework for burning objects. *Technical Report, Computer Science Department, Texas A & M University*, pages 2005–3–1, 2005.

[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.

[Ngu04] Hubert Nguyen. Fire in the "vulcan" demo. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 87–105. Addison-Wesley Professional, 2004.

[Owe05] John Owens. Streaming architectures and technology trends. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 457–470. Addison-Wesley Professional, 2005.

[Per85] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[Pla06] Planck law, 2006. [Online; accessed 17-May-2006]. `http://scienceworld.wolfram.com/physics/PlanckLaw.html`.

[PP94] C. H. Perry and R. W. Picard. Synthesizing flames and their spreading. In *Computer Animation and Simulation '94*, pages 1–14, 1994.

[Psz04] Dorota Pszczolkowska. Visual model of fire. In *EUROGRAPHICS 2004: 25th Annual Conference of the European Association for Computer Graphics*, pages 85–88. INRIA and the Eurographics Association, 2004.

[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. *ACM Trans. Graph.*, 22(3):703–707, 2003.

[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.

[SF95] Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 129–136, New York, NY, USA, 1995. ACM Press.

[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.

[Sta00] Jos Stam. Interacting with smoke and fire in real time. *Commun. ACM*, 43(7):76–83, 2000.

[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.

[YOH00] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins. Animating explosions. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 29–36, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[ZWF$^+$03] Ye Zhao, Xiaoming Wei, Zhe Fan, Arie Kaufman, and Hong Qin. Voxels on fire. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 36, Washington, DC, USA, 2003. IEEE Computer Society.

# Appendix A

# Glossary

| | |
|---|---|
| 2D slice simulation | A fire or fluid simulation performed in a set of 2D slices, implicitly defining a 3D volume by volumetric extrusion. |
| 3D simulation | A fire or fluid simulation performed in a 3D voxel grid. |
| CFD | Computational fluid dynamics. |
| Cg | NVIDIA's high-level language for GPU programming. |
| Cg fragment program | A fragment program written in Cg. |
| Cg vertex program | A vertex program written in Cg. |
| CG | Computer graphics. |
| Computational domain | The domain in which the behavior of a fluid is modeled. |
| CPU | Central processing unit. |
| Dynamic illumination | Illumination of the scene which varies and flickers due to variations in the fire. |
| Dynamic point light | A point light whose intensity varies over time based on the fire color field. |
| Fire color field | A field representing the fire with realistic colors. |
| Fire density fields | The scalar fuel gas, exhaust gas, and temperature fields used in the fire simulation. |
| Fire simulation | A simulation governing the behavior of a fire in a computational domain. |
| Fluid | A substance which flows under pressure. |
| Fluid simulation | A simulation governing the behavior of a fluid in motion. |
| FPS | Frames per second. |
| Fragment program | A program for performing operations on a stream of fragments on the GPU. |
| Frame rate | The frequency at which frames are rendered, usually measured in FPS. |
| GPU | Graphics processing unit. |
| Grid cell | A cell in the simulation grid corresponding to a discretized computational domain. |
| Grid dimensions | The width and height of a 2D simulation grid or additionally the depth of a 3D simulation grid. |
| LBM | Lattice Boltzmann Model. |
| Moving point light | A point light whose position varies over time. |

| | |
|---|---|
| OpenGL | An extensive API for 2D and 3D graphics. |
| Particle domain | The domain in which a particle simulation is defined. |
| Phong illumination model | A local illumination model for ambient, diffuse, and specular lighting. |
| Real-time | An application which runs above a given threshold frame rate (we define it to be 30 FPS). |
| Rendering pass | In our context, the drawing of a quad or a line to perform texture operations. |
| Simulation grid | The grid representing the computational domain in which the simulation is performed. |
| Smoke field | A field representing the smoke in the fire simulation. |
| Stable fluid solver | A stable method for performing fluid simulations at arbitrary timesteps. |
| Static point light | A point light whose intensity and position do not vary over time. |
| Stationary point light | A point light whose position is fixed and does not vary over time. |
| Texture operation | A GPU operation where a fragment program reads from and writes to textures. |
| Velocity field | A vector field specifying the direction and speed that air and gas move in. |
| Vertex program | A program for performing operations on a stream of vertices on the GPU. |
| Volumetric extrusion | An approach for implicitly defining a 3D volume from a set of 2D slices. |
| Vorticity confinement | An approach for increasing the vorticity and thus the turbulence of a velocity field. |
| Voxel grid | A 3D grid of uniform voxels. |

# Appendix B

# Application environment

This appendix describes the application environment for running our implementation of the fire simulation and visualization algorithms. We also describe a sample XML-file used for configuring the application, and finally we present all the console parameters which can be accessed during runtime.

## B.1 Running the program

The application executable *gpu-thesis.exe* can be found in the folder *application* of the accompanying ZIP-file. Running the application requires at least a graphics card with Pixel Shader 3.0 and Vertex Shader 3.0 functionality. The application takes exactly one argument, the filename of the XML-file specifying the application setup and parameters. The XML-file is described in the next section. A number of sample XML-files are included, as well as convenient bat-files for running them. Once the application has been started, the keys shown in table B.1 can be used for various purposes.

The console mentioned in the table can be used for interactive modification of simulation parameters and more. The main parameters which can be modified from the console are shown in section B.3. The console offers the command *help* to list all the console actions, and the command *list* to list all the modifiable parameters. Configuration files containing parameter values can be loaded and saved using the *load* and *save* commands respectively.

## B.2 XML-file

In this section we present the tags in the XML-file used when initializing the application. For each tag we specify the valid input type and for some tags a short description

| Keys | Purpose |
|---|---|
| w, a, s, d | Flying around in the scene. |
| i, j, k, l | Rotating the view. |
| Arrow keys | Moving the fire simulation around. |
| \| | Toggles the console. |
| t | Toggles the texture container which displays important textures. |
| c | Cycles the active texture set displayed in the texture container. |
| Number keys | Selects which textures to view in the texture container. |
| r, g, b | Select the active color channel in the texture container. |
| , | Ignite the fire simulation. |
| . | Reset the fire simulation. |
| p | Pause the fire simulation. |
| f | Toggle the FPS counter. |

Table B.1: Application keys

for clarification. The tags should be self-explanatory and for more information we refer to the ZIP-file where actual XML-files are found.

The `<fuel-sources>` tag is used to specify data used in the computation of the fuel source texture used in the fire simulation. For 2D fire slice simulation the image is scaled to fit the dimensions of one slice and each white pixel in the image specifies the location of fuel gas in each slice. When the fuel source texture is computed, for each white image pixel a random value between `<min-value>` and `<max-value>` is computed for the corresponding texel. For 3D fire simulation the source image is instead scaled to fit the width and depth of the 3D domain and each white image pixel specifies the location of fuel gas in the xz plane. The `<source-base-height>` tag specifies the first xz slice (counting from bottom to top along the y axis) to contain fuel sources, and the `<source-height>` tag specifies the number of xz slices to contain fuel sources (counting from bottom to top along the y axis). For each white image pixel and each xz slice a random value between `<min-value>` and `<max-value>` is computed and stored for the corresponding texel in the flat 3D texture.

The `<particle-sources>` tag are used in `<particle-system>`, `<smoke-system>`, and `<light-system>`, and specifies data used in the computation of the particle spawn positions. The spawn positions are computed in a similar way as the fuel source texture in the 3D case. The white image pixels specify the location of spawn positions in the xz plane, `<spawn-base-height>` specifies the first xz slice to contain spawn positions, `<spawn-height>` specifies the number of xz slices to contain spawn positions, and `<source-height>` corresponds to the height of the particle domain. For each particle a spawn position is randomly selected inside the volume defined by the tag values and scaled to fit into the particle domain.

```
<gpu-thesis>
    <setup>
        <config>string</config> // config filename
        <window>
            <fullscreen>true/false</fullscreen>
            <width>uint</width>
            <height>uint</height>
            <caption>string</caption> // window title
        </window>
        <camera>
            <origin>float float float</origin> // camera x y z position
            <pitch>float</pitch> // camera angle around the x-axis in degrees
            <yaw>float</yaw> // camera angle around the y-axis in degrees
        </camera>
        <fire-simulation>
            <slice>
                <count>uint</count> // number of slices in the slice simulation
            </slice>
            <full-3d>
                <computation-depth>uint</computation-depth>
                <source-base-height>uint</source-base-height>
                <source-height>uint</source-height>
            </full-3d>
            <computation-width>uint</computation-width>
            <computation-height>uint</computation-height>
            <velocity-boundary>
                <type>OPEN/SMOOTH/CONCRETE/ROUGH/WIND</type> // boundary type
            </velocity-boundary>
            <density-boundary>
                <type>OPEN/CLOSED</type> // boundary type
            </density-boundary>
            <fuel-sources>
                <image>string</image> // image file name
                <min-value>float</min-value> // minimum value in computed fuel source texture
                <max-value>float</max-value> // maximum value in computed fuel source texture
            </fuel-sources>
            <position>float float float</position> // x y z position of fire visualization
            <scale>float</scale> // to scale the extent of the fire visualization
            <visualization>
                <blackbody-radiation>
                    <tablesize>uint</tablesize> // number of entries in the lookup table
                    <min-temperature>double</min-temperature>
                    <max-temperature>double</max-temperature>
                    <exposure>double</exposure>
                </blackbody-radiation>
                <particle-system> // fire particle system
                    <particle-count>uint</particle-count>
                    <particle-texture>string</particle-texture> // image filename
                    <particle-texture-threshold>float</particle-texture-threshold>
                    // minimum particle texture texel intensity that should be visible
                    <render>true/false</render> // whether particles are rendered
                    <particle-sources>
                        <image>string</image> // image filename
                        <spawn-base-height>uint</spawn-base-height> // base height of spawn area
                        <spawn-height>uint</spawn-height> // height of spawn area
                        <source-height>uint</source-height> // total height of the spawn domain
                    </particle-sources>
                </particle-system>
                <smoke-system> // smoke particle system
                    <particle-count>uint</particle-count>
                    <particle-texture>string</particle-texture>
                    <particle-texture-threshold>float</particle-texture-threshold>
                    <render>true/false</render>
                    <particle-sources>
                        <image>string</image>
                        <spawn-base-height>uint</spawn-base-height>
                        <spawn-height>uint</spawn-height>
                        <source-height>uint</source-height>
                    </particle-sources>
                </smoke-system>
                <light-system> // light particle system
                    <particle-count>uint</particle-count> // number of moving lights
                    <color>float float float float</color> // base color for each light (r g b a)
                    <intensity>float</intensity> // total intensity of all lights
                    <render>true/false</render>
                    <particle-texture-threshold>float</particle-texture-threshold>
                    <particle-texture>string</particle-texture>
                    <particle-sources>
                        <image>string</image>
                        <spawn-base-height>uint</spawn-base-height>
                        <spawn-height>uint</spawn-height>
                        <source-height>uint</source-height>
                    </particle-sources>
                </light-system>
                <volume-renderer>
                    <type>SIMPLE/RAYMARCHER</type>
                </volume-renderer>
            </visualization>
        </fire-simulation>
        <scene>
            <global-ambient>float float float float</global-ambient> // r g b a
            <lights>
                <light>
                    <position>float float float</position> // x y z
                    <color>float float float float</color> // base color (r g b a)
                    <intensity>float</intensity> // base color multiplier
                    <dynamic>true/false</dynamic> // dynamic (true) or static (false)
                </light>
            </lights>
            <model-objects>
                <model-object>
                    <position>float float float</position> // x y z
                    <rotation-angle>float</rotation-angle> // angle in degrees
                    <rotation-axis>float float float</rotation-axis> // x y z
                    <scale>float</scale> // scale of model object
                    <relative>true/false</relative> // whether relative to the fire position
                    <filename>string</filename> // 3DS model filename
                    <ignore-meshes>string</ignore-meshes> // string of mesh indices
                    <face-normal-meshes>string</face-normal-meshes> // string of mesh indices
                    <material>
                        <ka>float</ka> // ambient
                        <kd>float</kd> // diffuse
                        <ks>float</ks> // specular
                        <shininess>float</shininess>
                    </material>
                </model-object>
            </model-objects>
            <noisy-surfaces>
                <noisy-surface>
                    <width>uint</width> // x resolution (decides triangle count)
                    <height>uint</height> // y resolution (decides triangle count)
                    <position>float float float</position> // x y z
                    <rotation-angle>float</rotation-angle> // angle in degrees
                    <rotation-axis>float float float</rotation-axis> // x y z
                    <scale>float</scale> // surface scale
                    <texture>string</texture> // image filename for surface texture
                    <amplitude>float</amplitude> // amplitude of noise
                    <frequency>float</frequency> // frequency of noise
```

```
                    <texture-scale>float</texture-scale> // scale of surface texture
                    <material>
                        <ka>float</ka> // ambient
                        <kd>float</kd> // diffuse
                        <ks>float</ks> // specular
                        <shininess>float</shininess>
                    </material>
                </noisy-surface>
            </noisy-surfaces>
        </scene>
    </setup>
    <timer>
        <runs>uint</runs> // number of runs
        <seconds>float</seconds> // duration of each run
        <log-file>string</log-file> // log filename
        <log-description>string</log-description> // description for element added to log file
    </timer>
</gpu-thesis>
```

## B.3 Console parameters

In table B.2 we present the most important parameters that can be updated in the console environment during the execution of the application.

| Parameter | Description |
|---|---|
| backgroundColor | The color in the framebuffer at the start of each frame. |
| consoleTransparency | The transparency of the console window. |
| fire.buoyancy | The buoyancy constant $f_b$ used in equation 4.2.13 to compute the strength of the buoyancy force. |
| fire.burningRate | The burning rate parameter $r$ used in equation 4.2.14 to compute the combustion parameter $C$. |
| fire.colorScale | The temperature scaling factor $T_{scale}$ used in equation 5.2.3 to control the brightness of the fire color computed. |
| fire.compute2dColors | Whether the fragment program `calculateColors2d()` is used instead of the fragment program `calculateColors()` to compute the fire colors. Both fragment programs are found in appendix D.3. |
| fire.exhaustDiffusion | The diffusion constant $\kappa_a$ used in equation 4.2.6 to compute the diffusion term. |
| fire.exhaustDissipation | The dissipation rate $\alpha_a$ used in equation 4.2.6 to compute the dissipation term. |
| fire.fuelDiffusion | The diffusion constant $\kappa_g$ used in equation 4.2.5 to compute the diffusion term. |
| fire.fuelDissipation | The dissipation rate $\alpha_g$ used in equation 4.2.5 to compute the dissipation term. |
| fire.globalWind | The x, y, and z components of the global wind vector used for the global wind boundary condition. |
| fire.globalWindAmplitude | The parameters $\gamma_{min}$, $\gamma_{max}$, and $\gamma_{frequency}$ used in equation 6.9.2 to compute the amplitude of the dynamic wind vector. |
| fire.globalWindAngle | The parameters $\alpha_{min}$, $\alpha_{max}$, and $\alpha_{frequency}$ used in equation 6.9.3 to compute the angle of the dynamic wind vector. |

| | |
|---|---|
| fire.globalWindDynamic | Whether the global wind vector should be dynamically updated based on two Perlin noise curves or not. |
| fire.gravity | The gravity constant $f_g$ used in equation 4.2.12 to compute the strength of the gravity force. |
| fire.jacobiSteps | The number of Jacobi steps used when computing the projection and diffusion parts of the fluid solver. |
| fire.massExpansion | The mass expansion rate $\phi$ used in equation 4.2.14 allowing the divergence of the velocity field to be non-zero. |
| fire.outputHeat | The parameter $T_0$ used in equation 4.2.17 to control the amount of heat that is produced by the combustion reaction. |
| fire.smokeColor | The parameter *smokeColor* used in the fragment program `calculateSmokeColors()` from appendix D.3 to set the color of the smoke. |
| fire.smokeIntensity | The parameter $s_{density}$ used in equation 5.3.1 to control the density of the smoke. |
| fire.smokeTemperatureScale | The parameter $T_{scale}$ used in equation 5.2.3 to scale the temperature to fit into the input range of the fall-off curve. |
| fire.smokeVisibilityThreshold | The amount of exhaust gas that must be present for the smoke to be visible (used in the fragment program `calculateSmokeColors()` found in appendix D.3). |
| fire.stoichiometricMixture | The stoichiometric mixture $b$ used in equation 4.2.14 to compute the combustion parameter $C$. |
| fire.temperatureDiffusion | The diffusion constant $\kappa_T$ used in equation 4.2.7 to compute the diffusion term. |
| fire.temperatureDissipation | The dissipation rate $\alpha_T$ used in equation 4.2.7 to compute the dissipation term. |
| fire.threshold | The ignition temperature of the fuel gas $T_{threshold}$ used in equation 4.2.14 to compute the combustion parameter $C$. |
| fire.vorticity | The parameter $\epsilon$ used in equation 4.2.10 to control the strength of the vorticity confinement force. |
| fire.vorticityWeight | The vorticity weight used to scale the vorticity force in the x and z directions where there is no combustion occurring (used in the fragment program `calculateVelocityForces()` from appendix D.3). |
| fixedTimestep | Whether the simulation is performed at a fixed timestep. |
| lightSystem.minimumLifeTime | The minimum lifetime of light particles (used in the fragment program `updatePositions()` from appendix D.4). |
| lightSystem.visibilityThreshold | The alpha threshold for respawning light particles (used in the fragment program `updatePositions()` from appendix D.4). |
| particleSystem.blendFunc | The blending approach used for fire. |
| particleSystem.drawBox | Whether the bounding box of the particle domain is displayed for debugging. |
| particleSystem.minimumLifeTime | The minimum lifetime of fire particles (used in the fragment program `updatePositions()` from appendix D.4). |
| particleSystem.particleSize | The size of the fire particles (used in the vertex program `renderParticles()` from appendix D.4). |
| particleSystem.visibilityThreshold | The alpha threshold for respawning fire particles (used in the fragment program `updatePositions()` from appendix D.4). |
| paused | Whether the simulation is paused. |

| | |
|---|---|
| scene.displayLights | Whether all point lights are displayed as small spheres for debugging. |
| showFps | Whether the FPS counter is shown on the screen. |
| smokeSystem.blendFunc | The blending approach used for smoke. |
| smokeSystem.minimumLifeTime | The minimum lifetime of smoke particles (used in the fragment program `updatePositions()` from appendix D.4). |
| smokeSystem.particleSize | The size of the smoke particles (used in the vertex program `renderParticles()` from appendix D.4). |
| smokeSystem.visibilityThreshold | The alpha threshold for respawning smoke particles (used in the fragment program `updatePositions()` from appendix D.4). |
| textureContainer.gridColor | The color of the grid overlaying each displayed texture. |
| textureContainer.separatorColor | The color of the grid surrounding all displayed textures. |
| textureContainer.show | Whether the texture container is displayed. |
| textureContainer.smoothDisplay | Whether the content of the displayed textures is interpolated when shown. |
| textureContainer.textColor | The color of the texture desriptions. |
| timestep | The timestep used when performing the simulation at a fixed timestep. |
| volume.boundingBox | The dimensions of the bounding box of the fire volume. |
| volume.drawBoundingBox | Whether the volume bounding box is displayed for debugging. |
| volume.stepSize | The distance between subsequent sample positions in the fire volume (used in equation 5.5.2). |

Table B.2: Description of configuration parameters.

# Appendix C

# Performance results

This appendix contains detailed performance tables for the performance results presented in section 7.1, as well as the configuration files used when producing the results. An explanation of the simulation parameters in the configuration files can be seen in appendix B.3.

## C.1 Performance tables

This section gives the performance tables. For each table we also include the central configuration parameters which were used to produce the results in the table. As mentioned in section 7.1, frame rates from three runs at 30 seconds each were averaged to produce the performance results. The hardware specifications and setup of the machine used to produce these results were shown in table 7.1.

| **Grid dimensions** | **Simulation only** | **Simulation and visualization** |
|---|---|---|
| 16x24x16 | 333.53 fps | 64.77 fps |
| 24x36x24 | 138.18 fps | 50.80 fps |
| 32x48x32 | 61.61 fps | 35.94 fps |
| 48x72x48 | 18.96 fps | 16.51 fps |
| 64x96x64 | 7.91 fps | 7.66 fps |

| **Number of voxels** | **Simulation only** |
|---|---|
| 6144 (16x24x16) | 3.00 ms |
| 20736 (24x36x24) | 7.24 ms |
| 49152 (32x48x32) | 16.23 ms |
| 165888 (48x72x48) | 52.76 ms |
| 393216 (64x96x64) | 126.45 ms |

| | |
|---|---|
| Window size | 1024x768 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Smoke particle count | 512 |
| Smoke particle size | 0.2 |
| Configuration file | fire-simulation-3d.cfg |

Table C.1: Top: Frame rates for 3D fire simulation with and without particle system visualization at various simulation grid dimensions. Middle: Processing time per frame for 3D fire simulation only as a function of the number of voxels in the grid.

| | **Slice count** | | | |
|---|---|---|---|---|
| **Grid dimensions** | **2** | **4** | **6** | **8** |
| 32x48 | 465.57 fps | 449.74 fps | 400.90 fps | 323.48 fps |
| 48x72 | 462.26 fps | 306.06 fps | 222.93 fps | 174.59 fps |
| 64x96 | 337.37 fps | 196.83 fps | 139.48 fps | 107.93 fps |
| 96x144 | 182.94 fps | 98.01 fps | 67.42 fps | 51.10 fps |
| 128x192 | 110.82 fps | 57.79 fps | 39.21 fps | 29.59 fps |
| 192x288 | 51.94 fps | 26.36 fps | 16.33 fps | 12.38 fps |
| 256x384 | 29.77 fps | 13.73 fps | 9.31 fps | 7.00 fps |

| | **Slice count** | | | |
|---|---|---|---|---|
| **Grid dimensions** | **2** | **4** | **6** | **8** |
| 32x48 | 2.15 ms | 2.22 ms | 2.49 ms | 3.09 ms |
| 48x72 | 2.16 ms | 3.27 ms | 4.49 ms | 5.73 ms |
| 64x96 | 2.96 ms | 5.08 ms | 7.17 ms | 9.27 ms |
| 96x144 | 5.47 ms | 10.20 ms | 14.83 ms | 19.57 ms |
| 128x192 | 9.02 ms | 17.30 ms | 25.50 ms | 33.80 ms |
| 192x288 | 19.25 ms | 37.93 ms | 61.23 ms | 80.78 ms |
| 256x384 | 33.59 ms | 72.82 ms | 107.42 ms | 142.94 ms |

| | |
|---|---|
| Window size | 1024x768 |
| Configuration file | fire-simulation-slice.cfg |

Table C.2: Top: Frame rates for 2D fire slice simulation only at various simulation grid dimensions and slice counts. Middle: Processing time per frame for 2D fire slice simulation only at various simulation grid dimensions and slice counts.

| | **Slice count** | | | |
|---|---|---|---|---|
| **Grid dimensions** | **2** | **4** | **6** | **8** |
| 32x48 | 71.17 fps | 68.82 fps | 65.94 fps | 63.76 fps |
| 48x72 | 67.56 fps | 62.52 fps | 58.10 fps | 54.48 fps |
| 64x96 | 63.69 fps | 56.26 fps | 50.53 fps | 45.72 fps |
| 96x144 | 55.01 fps | 43.99 fps | 36.69 fps | 31.34 fps |
| 128x192 | 46.23 fps | 33.73 fps | 26.50 fps | 21.77 fps |
| 192x288 | 31.70 fps | 20.06 fps | 14.82 fps | 11.68 fps |
| 256x384 | 21.59 fps | 12.79 fps | 9.00 fps | 6.88 fps |

| | |
|---|---|
| Window size | 1024x768 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Smoke particle count | 512 |
| Smoke particle size | 0.2 |
| Configuration file | fire-simulation-slice.cfg |

Table C.3: Frame rates for 2D fire slice simulation with particle system visualization at various simulation grid dimensions and slice counts.

| | | Dynamic timestep | | Fixed timestep |
|---|---|---|---|---|
| **Particle count** | **Particle size** | **No rendering** | **Rendering** | **Rendering** |
| 256 | 0.1 | 332.02 fps | 195.06 fps | 389.78 fps |
| 512 | 0.08 | 330.95 fps | 142.57 fps | 227.02 fps |
| 1024 | 0.06 | 331.35 fps | 100.88 fps | 138.07 fps |
| 2048 | 0.04 | 321.21 fps | 91.47 fps | 121.87 fps |
| 4096 | 0.035 | 318.22 fps | 66.34 fps | 81.86 fps |
| 8192 | 0.025 | 298.38 fps | 61.78 fps | 75.73 fps |

| | Rendering | |
|---|---|---|
| **Particle count** | **Dynamic timestep** | **Fixed timestep** |
| 256 | 5.13 ms | 2.57 ms |
| 512 | 7.01 ms | 4.40 ms |
| 1024 | 9.91 ms | 7.24 ms |
| 2048 | 10.93 ms | 8.21 ms |
| 4096 | 15.07 ms | 12.22 ms |
| 8192 | 16.19 ms | 13.20 ms |

| | |
|---|---|
| Window size | 1024x768 |
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | fire-simulation-slice.cfg |

Table C.4: Top: Frame rates for particle system visualization of fire at various fire particle counts. Middle: Processing time per frame for particle system visualization of fire at various fire particle counts.

| | | Dynamic timestep | | Fixed timestep |
|---|---|---|---|---|
| **Particle count** | **Particle size** | **No rendering** | **Rendering** | **Rendering** |
| 64 | 0.3 | 91.52 fps | 83.10 fps | 107.86 fps |
| 128 | 0.26 | 91.53 fps | 77.85 fps | 98.81 fps |
| 256 | 0.24 | 91.21 fps | 69.45 fps | 86.00 fps |
| 512 | 0.2 | 91.15 fps | 59.13 fps | 70.60 fps |
| 1024 | 0.14 | 90.99 fps | 47.26 fps | 54.91 fps |
| 2048 | 0.1 | 90.46 fps | 31.14 fps | 34.32 fps |

| | |
|---|---|
| Window size | 1024x768 |
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | fire-simulation-slice.cfg |

Table C.5: Frame rates for particle system visualization of fire and smoke at various smoke particle counts.

| Grid dimensions | Dynamic timestep | Fixed timestep |
|---|---|---|
| 16x24x16 | 31.60 fps | 30.94 fps |
| 24x36x24 | 19.59 fps | 16.69 fps |
| 32x48x32 | 13.53 fps | 8.46 fps |
| 48x72x48 | 7.19 fps | 3.54 fps |
| 64x96x64 | 4.12 fps | 1.76 fps |

| | |
|---|---|
| Window size | 1024x768 |
| Step size | 1.0 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | volume.cfg |

Table C.6: Frame rates for volume rendering the fire with no smoke using a 3D fire simulation at various grid dimensions.

| Step size | Dynamic timestep | Fixed timestep |
|---|---|---|
| 0.2 | 4.54 fps | 4.27 fps |
| 0.4 | 8.90 fps | 7.47 fps |
| 0.6 | 12.57 fps | 10.41 fps |
| 0.8 | 16.41 fps | 13.77 fps |
| 1.0 | 19.98 fps | 16.65 fps |
| 1.2 | 22.76 fps | 20.23 fps |

| | |
|---|---|
| Window size | 1024x768 |
| Grid dimensions | 24x36x24 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | volume.cfg |

Table C.7: Frame rates for volume rendering the fire with no smoke at various step sizes using a 3D fire simulation at 24x36x24.

| Light count | Fixed lights | Moving lights |
|---|---|---|
| 1 | 59.65 fps | 58.04 fps |
| 2 | 53.66 fps | 52.87 fps |
| 4 | 44.78 fps | 44.52 fps |
| 8 | 33.39 fps | 33.16 fps |
| 16 | 22.12 fps | 22.02 fps |
| 32 | 13.92 fps | 14.01 fps |
| 64 | 7.59 fps | 7.72 fps |
| 128 | 4.01 fps | 3.94 fps |

| | |
|---|---|
| Window size | 1024x768 |
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Smoke particle count | 512 |
| Smoke particle size | 0.2 |
| Configuration file | fire-simulation-slice.cfg |

Table C.8: Frame rates for dynamic illumination with stationary and moving dynamic lights at various light counts using a dynamic timestep.

| **Light count** | **Fixed lights** | **Moving lights** |
|---|---|---|
| 1 | 64.78 fps | 64.06 fps |
| 2 | 56.90 fps | 57.15 fps |
| 4 | 46.40 fps | 46.29 fps |
| 8 | 33.94 fps | 33.51 fps |
| 16 | 21.64 fps | 21.53 fps |
| 32 | 12.64 fps | 12.56 fps |
| 64 | 7.03 fps | 7.13 fps |
| 128 | 3.85 fps | 3.79 fps |

| **Light count** | **Moving lights** |
|---|---|
| 1 | 15.61 ms |
| 2 | 17.50 ms |
| 4 | 21.61 ms |
| 8 | 29.84 ms |
| 16 | 46.45 ms |
| 32 | 79.64 ms |
| 64 | 140.21 ms |
| 128 | 264.13 ms |

| | |
|---|---|
| Window size | 1024x768 |
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Smoke particle count | 512 |
| Smoke particle size | 0.2 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | fire-simulation-slice.cfg |

Table C.9: Top: Frame rates for dynamic illumination with stationary and moving dynamic lights at various light counts using a fixed timestep. Middle: Processing time per frame for dynamic illumination with moving dynamic lights at various light counts using a fixed timestep.

| **Window dimensions** | **Simulation and visualization** |
|---|---|
| 640x480 | 76.40 fps |
| 800x600 | 62.76 fps |
| 1024x768 | 46.43 fps |
| 1280x960 | 39.54 fps |

| | |
|---|---|
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Light count (stationary) | 4 |
| Fire particle count | 2048 |
| Fire particle size | 0.04 |
| Smoke particle count | 512 |
| Smoke particle size | 0.2 |
| Timestep | $\frac{1}{30}$ sec |
| Configuration file | fire-simulation-slice.cfg |

Table C.10: Frame rates for simulation and visualization of fire and smoke with dynamic illumination at various window dimensions.

| **Window dimensions** | **Simulation only** | **Simulation only (no flip)** |
|---|---|---|
| 640x480 | 373.28 fps | 400.80 fps |
| 800x600 | 358.40 fps | 399.38 fps |
| 1024x768 | 337.92 fps | 397.06 fps |
| 1280x960 | 312.28 fps | 397.31 fps |

| | |
|---|---|
| Grid dimensions | 64x96 |
| Slice count | 2 |
| Configuration file | fire-simulation-slice.cfg |

Table C.11: Frame rates for simulation only at various window dimensions. The *no flip* column corresponds to a modified version of the program were SDL_GL_SwapBuffers() was deactivated.

## C.2 Configuration files

Here we show the configuration files used for the results presented in the previous section.

### C.2.1 fire-simulation-3d.cfg

```
backgroundColor 0.4 0.45 0.5
fire.buoyancy 0.08
fire.burningRate 8
fire.colorScale 0.052
fire.compute2dColors 0
fire.exhaustDiffusion 5
fire.exhaustDissipation 2.2
fire.fuelDiffusion 10
fire.fuelDissipation 0.5
fire.globalWind 0 0 0
fire.globalWindAmplitude 0 0 0
fire.globalWindAngle 0 0 0
fire.globalWindDynamic 0
fire.gravity 0.01
fire.jacobiSteps 20
fire.massExpansion 0
fire.outputHeat 3000
fire.smokeColor 0.33 0.3 0.3
fire.smokeIntensity 0.2
fire.smokeTemperatureScale 0.0012
fire.smokeVisibilityThreshold 0.02
fire.stoichiometricMixture 0.6
fire.temperatureDiffusion 10
fire.temperatureDissipation 2.2
fire.threshold 250
fire.vorticity 20
fire.vorticityWeight 1
fixedTimestep 0
particleSystem.blendFunc 3
particleSystem.drawBox 0
particleSystem.minimumLifeTime 0.5
particleSystem.particleSize 0.04
particleSystem.visibilityThreshold 0.02
paused 0
showFps 0
smokeSystem.blendFunc 5
smokeSystem.minimumLifeTime 0.5
smokeSystem.particleSize 0.2
smokeSystem.visibilityThreshold 0.004
timestep 0.0333333
```

### C.2.2 fire-simulation-slice.cfg

```
backgroundColor 0.4 0.45 0.5
fire.buoyancy 0.08
fire.burningRate 8
fire.colorScale 0.052
fire.compute2dColors 0
fire.exhaustDiffusion 5
fire.exhaustDissipation 3.1
fire.fuelDiffusion 10
fire.fuelDissipation 0.5
fire.globalWind 0 0 0
fire.globalWindAmplitude 0 0 0
fire.globalWindAngle 0 0 0
fire.globalWindDynamic 0
fire.gravity 0.01
fire.jacobiSteps 20
fire.massExpansion 0
fire.outputHeat 3000
fire.smokeColor 0.33 0.3 0.3
fire.smokeIntensity 0.2
fire.smokeTemperatureScale 0.001
fire.smokeVisibilityThreshold 0.02
fire.stoichiometricMixture 0.6
fire.temperatureDiffusion 10
fire.temperatureDissipation 2.9
fire.threshold 250
fire.vorticity 42
fire.vorticityWeight 1
fixedTimestep ?
particleSystem.blendFunc 3
particleSystem.drawBox 0
particleSystem.minimumLifeTime 0.5
particleSystem.particleSize ?
particleSystem.visibilityThreshold 0.02
paused 0
showFps 0
smokeSystem.blendFunc 5
smokeSystem.minimumLifeTime 0.5
smokeSystem.particleSize ?
smokeSystem.visibilityThreshold 0.004
timestep 0.0333333
```

### C.2.3 volume.cfg

```
backgroundColor 0.05 0.1 0.1
fire.buoyancy 0.02
fire.burningRate 1.25
fire.colorScale 0.008
fire.compute2dColors 0
fire.exhaustDiffusion 0.01
fire.exhaustDissipation 40
fire.fuelDiffusion 0.01
fire.fuelDissipation 0.5
fire.globalWind 14 0 0
fire.globalWindAmplitude 0 20 3.5
fire.globalWindAngle 0 540 1
fire.globalWindDynamic 1
fire.gravity 0.01
fire.jacobiSteps 20
fire.massExpansion 1
fire.outputHeat 100000
fire.smokeColor 0.5 0.4 0.4
fire.smokeIntensity 0.2
fire.smokeTemperatureScale 8e-007
fire.smokeVisibilityThreshold 0.02
fire.stoichiometricMixture 0.6
fire.temperatureDiffusion 0.01
fire.temperatureDissipation 8
fire.threshold 250
fire.vorticity 40
fire.vorticityWeight 1
```

```
fixedTimestep 1
paused 0
showFps 0
timestep 0.0333333
volume.boundingBox -0.5 0 -0.5 0.5 1 0.5
volume.drawBoundingBox 0
volume.stepSize ?
```

# Appendix D

# Cg fragment and vertex programs

Here we show all the Cg programs used for fire simulation and visualization. Some of these programs use utility Cg functions shown in section D.7.

## D.1 Fluid 3D

```
#include "util.cg"

void advect3d(float2 coords : TEXCOORD0,        // grid coordinates
              out float4 result : COLOR,        // advected quantity
              uniform float timestep,           // time since last frame
              uniform float voxelSize,          // size (width/height/depth) of a single voxel
              uniform float4 uniformAdvect,     // uniform advect vector
              uniform samplerRECT velocity,     // input velocity field
              uniform samplerRECT value,        // quantity to advect
              uniform samplerRECT offsetLookup, // offset lookup table
              uniform samplerRECT domainLookup, // domain lookup table
              uniform float width,              // domain width
              uniform float height,             // domain height
              uniform float depth)              // domain depth
{
    float3 localCoords = texRECT(domainLookup, coords).xyz;

    float3 localVelocity = texRECT(velocity, coords).xyz + uniformAdvect.xyz / timestep;

    float3 invVelocity = 1 / localVelocity;

    float t = timestep * voxelSize;

    float3 t0 = (localCoords - 0.5) * invVelocity;
    float3 t1 = (localCoords - float3(width, height, depth) - 0.5) * invVelocity;

    // clamp stepsize to local domain
    if (t0.x >= 0) {
        if (t0.x < t)
            t = t0.x;
    } else if (t1.x >= 0 && t1.x < t)
        t = t1.x;

    if (t0.y >= 0) {
        if (t0.y < t)
            t = t0.y;
    } else if (t1.y >= 0 && t1.y < t)
        t = t1.y;

    if (t0.z >= 0) {
        if (t0.z < t)
            t = t0.z;
    } else if (t1.z >= 0 && t1.z < t)
        t = t1.z;

    // calculate old location, where we would end up by going back a timestep
    float3 oldValueLocation = localCoords - localVelocity * t;

    // sample value using interpolation
    result = lookup3d(oldValueLocation, value,
                      offsetLookup, float3(width, height, depth));
}


void addForce3d(float2 coords : TEXCOORD0,        // grid coordinates
                out float4 result : COLOR,        // advected quantity
                uniform float timestep,           // time since last frame
                uniform samplerRECT force,        // input force field
                uniform samplerRECT value)        // quantity to add force to
{
    result = texRECT(value, coords) + texRECT(force, coords) * timestep;
}


void jacobi3d(float2 coords : TEXCOORD0,            // grid coordinates
              out float4 result : COLOR,            // resulting value
              uniform float4 alpha,                 // alpha
              uniform float4 rBeta,                 // reciprocal beta
              uniform samplerRECT x,                // x vector (Ax = b)
              uniform samplerRECT b,                // b vector (ax = b)
              uniform samplerRECT offsetLookup,     // offset lookup table
              uniform samplerRECT domainLookup,     // domain lookup table
              uniform samplerRECT neighborLookup)   // neighbor lookup table
{
    // left, right, bottom, top, next slice and previous slice samples

    float4 neighbors = f4texRECT(neighborLookup, coords);

    float4 xLeft     = f4texRECT(x, coords - float2(1, 0));
    float4 xRight    = f4texRECT(x, coords + float2(1, 0));
    float4 xBottom   = f4texRECT(x, coords - float2(0, 1));
    float4 xTop      = f4texRECT(x, coords + float2(0, 1));

    float4 xNext     = f4texRECT(x, neighbors.xy);
    float4 xPrevious = f4texRECT(x, neighbors.zw);

    // b sample, from center
    float4 bCenter = f4texRECT(b, coords);

    // evaluate jacobi iteration
    result = (alpha * (xLeft + xRight + xBottom + xTop + xNext + xPrevious) +
              bCenter) * rBeta;
}
```

```
void divergence3d(float2 coords : TEXCOORD0,         // grid coordinates
                  out float4 result : COLOR,         // divergence
                  uniform float halfVoxelSize,       // voxelSize / 2
                  uniform samplerRECT velocity,      // velocity field
                  uniform samplerRECT offsetLookup,  // local offset lookup table
                  uniform samplerRECT domainLookup)  // local domain lookup table
{
    float4 vLeft     = f4texRECT(velocity, coords - float2(1, 0));
    float4 vRight    = f4texRECT(velocity, coords + float2(1, 0));
    float4 vBottom   = f4texRECT(velocity, coords - float2(0, 1));
    float4 vTop      = f4texRECT(velocity, coords + float2(0, 1));

    float3 localCoords = f4texRECT(domainLookup, coords).xyz;

    float4 vNext     = f4texRECT(velocity,
                                 localCoords.xy + f2texRECT(offsetLookup,
                                                            localCoords.z + 1));
    float4 vPrevious = f4texRECT(velocity,
                                 localCoords.xy + f2texRECT(offsetLookup,
                                                            localCoords.z - 1));

    result = -halfVoxelSize * ((vRight.x - vLeft.x) +
                               (vTop.y - vBottom.y) +
                               (vNext.z - vPrevious.z));
}


void subtractGradient3d(float2 coords : TEXCOORD0,        // grid coordinates
                        out float4 result : COLOR,        // new velocity
                        uniform float halfVoxelSize,      // voxelSize / 2
                        uniform samplerRECT pressure,     // pressure field
                        uniform samplerRECT velocity,     // velocity field
                        uniform samplerRECT offsetLookup, // offset lookup table
                        uniform samplerRECT domainLookup) // domain lookup table
{
    // left, right, bottom, top, next slice and previous slice samples
    float pLeft   = f1texRECT(pressure, coords - float2(1, 0));
    float pRight  = f1texRECT(pressure, coords + float2(1, 0));
    float pBottom = f1texRECT(pressure, coords - float2(0, 1));
    float pTop    = f1texRECT(pressure, coords + float2(0, 1));

    float3 localCoords = f4texRECT(domainLookup, coords).xyz;

    float pPrevious = f1texRECT(pressure,
                                localCoords.xy + f2texRECT(offsetLookup,
                                                           localCoords.z - 1));
    float pNext     = f1texRECT(pressure,
                                localCoords.xy + f2texRECT(offsetLookup,
                                                           localCoords.z + 1));

    result = f4texRECT(velocity, coords);

    // compute and subtract gradient
    result.xyz -= 0.5 * float3(pRight - pLeft,
                               pTop - pBottom,
                               pNext - pPrevious) / (2 * halfVoxelSize);
}


void boundary3d(float2 coords : TEXCOORD0,        // boundary coordinates
                float3 offset : TEXCOORD1,        // offset coordinates
                out float4 result : COLOR,        // resulting boundary values
                uniform float scale,              // parameter for different boundary conditions
                uniform samplerRECT value,        // input field
                uniform samplerRECT offsetLookup, // offset lookup table
                uniform samplerRECT domainLookup) // domain lookup table

{
    if (abs(offset.z) > 0) { // z-boundary
        float3 localCoords = f4texRECT(domainLookup, coords).xyz;
        float2 globalCoords = localCoords.xy + f2texRECT(offsetLookup,
                                                localCoords.z + offset.z);
        result = scale * f4texRECT(value, globalCoords);
    } else {
        result = scale * f4texRECT(value, coords + offset.xy);
    }
}


void boundaryOpen(float2 coords : TEXCOORD0,        // boundary coordinates
                  float2 offset : TEXCOORD1,        // offset coordinates
                  out float4 result : COLOR,        // resulting boundary values
                  uniform float4 v,                 // boundary condition parameter
                  uniform samplerRECT value)        // input field
{
    result = v;
}


void boundaryVelocity3d(float2 coords : TEXCOORD0,        // boundary coordinates
                        float3 offset : TEXCOORD1,        // offset coordinates
                        out float4 result : COLOR,        // resulting boundary values
                        uniform float normScale,          // normal boundary scale
                        uniform float perpScale,          // perpendicular boundary scale
                        uniform samplerRECT value,        // velocity field
                        uniform samplerRECT offsetLookup, // offset lookup table
                        uniform samplerRECT domainLookup) // domain lookup table

{
    float3 localCoords = f4texRECT(domainLookup, coords + offset.xy).xyz;
    float2 globalCoords = localCoords.xy +
                          f2texRECT(offsetLookup, localCoords.z + offset.z);
    result = f4texRECT(value, globalCoords);

    if (abs(offset.x) > 0) // right or left
        result *= float4(normScale, perpScale, perpScale, 0);
    else if (abs(offset.y) > 0) // top or bottom
        result *= float4(perpScale, normScale, perpScale, 0);
    else
        result *= float4(perpScale, perpScale, normScale, 0);
}


void boundaryWind3d(out float4 result : COLOR, // resulting boundary values
                    uniform float4 globalWind) // global wind
{
    result = globalWind;
}


void generateVorticity3d(float2 coords : TEXCOORD0,       // grid coordinates
                       out float4 result : COLOR,         // resulting vorticity
                       uniform float halfVoxelSize,       // voxelSize / 2
                       uniform samplerRECT velocity,      // velocity field
                       uniform samplerRECT offsetLookup,  // offset lookup table
                       uniform samplerRECT domainLookup)  // domain lookup table
```

```
{
    float4 vLeft     = f4texRECT(velocity, coords - float2(1, 0));
    float4 vRight    = f4texRECT(velocity, coords + float2(1, 0));
    float4 vBottom   = f4texRECT(velocity, coords - float2(0, 1));
    float4 vTop      = f4texRECT(velocity, coords + float2(0, 1));

    float3 localCoords = f4texRECT(domainLookup, coords).xyz;

    float4 vNext     = f4texRECT(velocity, localCoords.xy +
                                    f2texRECT(offsetLookup, localCoords.z + 1));
    float4 vPrevious = f4texRECT(velocity, localCoords.xy +
                                    f2texRECT(offsetLookup, localCoords.z - 1));

    float4 dvdx = halfVoxelSize * (vRight - vLeft);
    float4 dvdy = halfVoxelSize * (vTop - vBottom);
    float4 dvdz = halfVoxelSize * (vNext - vPrevious);

    result = float4((dvdy.z - dvdz.y), (dvdz.x - dvdx.z), (dvdx.y - dvdy.x), 1);
}


void addVorticity3d(float2 coords : TEXCOORD0,        // grid coordinates
                    out float4 result : COLOR,        // resulting velocity force
                    uniform float halfVoxelSize,      // voxelSize / 2
                    uniform float epsilon,            // vorticity scale
                    uniform samplerRECT vorticity,    // vorticity field
                    uniform samplerRECT offsetLookup, // offset lookup table
                    uniform samplerRECT domainLookup) // domain lookup table
{
    float3 local = f3texRECT(vorticity, coords);

    float wLeft     = length(f3texRECT(vorticity, coords - float2(1, 0)));
    float wRight    = length(f3texRECT(vorticity, coords + float2(1, 0)));
    float wBottom   = length(f3texRECT(vorticity, coords - float2(0, 1)));
    float wTop      = length(f3texRECT(vorticity, coords + float2(0, 1)));

    float3 localCoords = f4texRECT(domainLookup, coords).xyz;

    float wNext     = length(f3texRECT(vorticity, localCoords.xy +
                                          f2texRECT(offsetLookup,
                                                    localCoords.z + 1)));

    float wPrevious = length(f3texRECT(vorticity, localCoords.xy +
                                          f2texRECT(offsetLookup,
                                                    localCoords.z - 1)));

    float3 wGrad = halfVoxelSize * float3(wRight - wLeft,
                                          wTop - wBottom,
                                          wNext - wPrevious);

    float wGradLength = length(wGrad);

    if (wGradLength > 0) {
        float3 normal = normalize(wGrad);

        result.xyz = epsilon * halfVoxelSize * 2 * (normal.yzx * local.zxy -
                                                     normal.zxy * local.yzx);
        result.w = 0;
    } else
        result = float4(0, 0, 0, 0);
}
```

## D.2 Fluid 2D

```
#include "util.cg"

void advectSlice(float2 coords : TEXCOORD0,        // grid coordinates
                 out float4 result : COLOR,        // advected quantity
                 uniform float timestep,           // time since last frame
                 uniform float voxelSize,          // width/height/depth of a single voxel
                 uniform samplerRECT velocity,     // input velocity field
                 uniform samplerRECT value,        // quantity to advect
                 uniform samplerRECT domainLookup, // domain lookup table
                 uniform float width,              // domain width
                 uniform float height)             // domain height
{
    float2 localCoords = f2texRECT(domainLookup, coords);

    float2 localVelocity = f2texRECT(velocity, coords);

    float2 invVelocity = 1 / localVelocity;

    float t = timestep * voxelSize;

    float2 t0 = (localCoords - 0.5) * invVelocity;
    float2 t1 = (localCoords - (float2(width, height) - 0.5)) * invVelocity;

    // clamp stepsize to local domain
    if (t0.x >= 0) {
        if (t0.x < t)
            t = t0.x;
    } else if (t1.x >= 0 && t1.x < t)
        t = t1.x;

    if (t0.y >= 0) {
        if (t0.y < t)
            t = t0.y;
    } else if (t1.y >= 0 && t1.y < t)
        t = t1.y;

    // calculate old location, where we would end up by going back a timestep
    float2 oldValueLocation = coords - localVelocity * t;

    // interpolate grid values
    result = f4texRECTbilerp(value, oldValueLocation);
}


void addForce2d(float2 coords : TEXCOORD0,        // grid coordinates
                out float4 result : COLOR,        // advected quantity
                uniform float timestep,           // time since last frame
                uniform samplerRECT force,        // input force field
                uniform samplerRECT value)        // quantity to add force to
{
    result = texRECT(value, coords) + texRECT(force, coords) * timestep;
}


void jacobi2d(float2 coords : TEXCOORD0,        // grid coordinates
              out float4 result : COLOR,        // resulting value
              uniform float4 alpha,             // alpha
              uniform float4 rBeta,             // reciprocal beta
              uniform samplerRECT x,            // x vector (Ax = b)
              uniform samplerRECT b)            // b vector (ax = b)
{
    // left, right, bottom and top slice samples
    float4 xLeft     = f4texRECT(x, coords - float2(1, 0));
    float4 xRight    = f4texRECT(x, coords + float2(1, 0));
    float4 xBottom   = f4texRECT(x, coords - float2(0, 1));
    float4 xTop      = f4texRECT(x, coords + float2(0, 1));

    // b sample, from center
    float4 bCenter = f4texRECT(b, coords);

    // evaluate jacobi iteration
    result = (alpha * (xLeft + xRight + xBottom + xTop) + bCenter) * rBeta;
}


void divergence2d(float2 coords : TEXCOORD0,         // grid coordinates
                  out float4 result : COLOR,         // divergence
                  uniform float halfVoxelSize,       // voxelSize / 2
                  uniform samplerRECT velocity)      // velocity field
{
    float4 vLeft     = f4texRECT(velocity, coords - float2(1, 0));
    float4 vRight    = f4texRECT(velocity, coords + float2(1, 0));
    float4 vBottom   = f4texRECT(velocity, coords - float2(0, 1));
    float4 vTop      = f4texRECT(velocity, coords + float2(0, 1));

    result = -halfVoxelSize * ((vRight.x - vLeft.x) + (vTop.y - vBottom.y));
}


void subtractGradient2d(float2 coords : TEXCOORD0,    // grid coordinates
                        out float4 result : COLOR,    // new velocity
                        uniform float halfVoxelSize,  // voxelSize / 2
                        uniform samplerRECT pressure, // pressure field
                        uniform samplerRECT velocity) // velocity field
{
    // left, right, bottom and top slice samples
    float pLeft   = f1texRECT(pressure, coords - float2(1, 0));
    float pRight  = f1texRECT(pressure, coords + float2(1, 0));
    float pBottom = f1texRECT(pressure, coords - float2(0, 1));
    float pTop    = f1texRECT(pressure, coords + float2(0, 1));

    result = f4texRECT(velocity, coords);

    // compute and subtract gradient
    result.xy -= 0.5 * float2(pRight - pLeft, pTop - pBottom) / (2 * halfVoxelSize);
}


void boundary2d(float2 coords : TEXCOORD0,        // boundary coordinates
                float2 offset : TEXCOORD1,        // offset coordinates
                out float4 result : COLOR,        // resulting boundary values
                uniform float scale,              // parameter for different boundary conditions
                uniform samplerRECT value)        // input field
{
    result = scale * f4texRECT(value, coords + offset);
}


void boundaryOpen(float2 coords : TEXCOORD0,        // boundary coordinates
                  float2 offset : TEXCOORD1,        // offset coordinates
                  out float4 result : COLOR,        // resulting boundary values
                  uniform float4 v,                 // parameter for different boundary conditions
                  uniform samplerRECT value)        // input field
{
    result = v;
```

```
}


void boundaryVelocity2d(float2 coords : TEXCOORD0,  // boundary coordinates
                        float2 offset : TEXCOORD1,  // offset coordinates
                        out float4 result : COLOR,  // resulting boundary values
                        uniform float normScale,    // normal boundary scale
                        uniform float perpScale,    // perpendicular boundary scale
                        uniform samplerRECT value)  // velocity field
{
    result = f4texRECT(value, coords + offset);

    if (abs(offset.x) > 0) { // right or left
        result *= float4(normScale, perpScale, 0, 0);
    } else { // top or bottom
        result *= float4(perpScale, normScale, 0, 0);
    }
}


void boundaryVelocityOpen2d(float2 coords : TEXCOORD0, // boundary coordinates
                            float2 offset : TEXCOORD1, // offset coordinates
                            out float4 result : COLOR, // resulting boundary values
                            uniform float scale,       // boundary condition parameter
                            uniform samplerRECT value) // velocity field
{
    result = 0.5 * scale * f4texRECT(value, coords + offset);
}


void boundaryWindSlice(float2 coords : TEXCOORD0,        // boundary coordinates
                       out float4 result : COLOR,        // resulting boundary values
                       uniform float4 globalWind,        // global wind vector
                       uniform float numSlices,          // slice count
                       uniform samplerRECT domainLookup) // domain lookup table
{
    float4 globalCoords = f4texRECT(domainLookup, coords);

    float theta = (globalCoords.z - 0.5) * 3.14159 / numSlices;
    float2 sliceVector = float2(cos(theta), sin(theta));
    float cosTheta = dot(sliceVector, globalWind.xz);

    result = float4(cosTheta * length(globalWind.xz), globalWind.y, 0, 0);
}


void generateVorticity2d(float2 coords : TEXCOORD0,     // grid coordinates
                         out float4 result : COLOR,     // resulting vorticity
                         uniform float halfVoxelSize,   // voxelSize / 2
                         uniform samplerRECT velocity)  // velocity field
{
    float4 vLeft     = f4texRECT(velocity, coords - float2(1, 0));
    float4 vRight    = f4texRECT(velocity, coords + float2(1, 0));
    float4 vBottom   = f4texRECT(velocity, coords - float2(0, 1));
    float4 vTop      = f4texRECT(velocity, coords + float2(0, 1));

    float dvdx = halfVoxelSize * (vRight.y - vLeft.y);
    float dvdy = halfVoxelSize * (vTop.x - vBottom.x);

    // w = grad cross velocity
    // lenght(w) saved in alpha channel
    result = float4(dvdx - dvdy, abs(dvdx - dvdy), 0, 0);
}
```

```
void addVorticity2d(float2 coords : TEXCOORD0,     // grid coordinates
                    out float4 result : COLOR,     // resulting velocity force
                    uniform float halfVoxelSize,   // voxelSize / 2
                    uniform float epsilon,         // vorticity scale
                    uniform samplerRECT vorticity) // vorticity field
{
    float2 local = f2texRECT(vorticity, coords);

    float wLeft     = f2texRECT(vorticity, coords - float2(1, 0)).y;
    float wRight    = f2texRECT(vorticity, coords + float2(1, 0)).y;
    float wBottom   = f2texRECT(vorticity, coords - float2(0, 1)).y;
    float wTop      = f2texRECT(vorticity, coords + float2(0, 1)).y;

    float2 wGrad = halfVoxelSize * float2(wRight - wLeft, wTop - wBottom);

    float wGradLength = length(wGrad);

    result = float4(0, 0, 0, 0);

    if (wGradLength > 0) {
        float2 normal = normalize(wGrad);

        // f_vorticity = epsilon * h * (N cross w)
        result.xy = epsilon * halfVoxelSize * 2 * float2(normal.y * local.x, -normal.x * local.x);
    }
}
```

## D.3 Fire

```
#include "util.cg"

void calculateDensityForces(float2 coords : TEXCOORD0,           // grid coordinates
                            out float4 result : COLOR,           // the resulting force
                            uniform float burningRate,           // burning rate parameter
                            uniform float outputHeat,            // reaction output heat
                            uniform float stoichiometricMixture, // oxygen mixture rate
                            uniform float threshold,             // reaction threshold
                            uniform float4 dissipation,          // dissipation rates
                            uniform samplerRECT densityField,    // density field
                            uniform samplerRECT sourceField)     // fuel sources
{
    float4 density = f4texRECT(densityField, coords);

    // add source field
    result = f4texRECT(sourceField, coords);

    // dissipate
    result -= dissipation * density;

    // combust
    float exhaust = density.x;
    float fuel = density.y;
    float temperature = density.z;

    if (temperature > threshold) {
        float C = burningRate * stoichiometricMixture * fuel;

        result.x += C * (1 + 1 / stoichiometricMixture); // add exhaust
        result.y += -C / stoichiometricMixture; // burn fuel
        result.z += outputHeat * C; // increase heat
    }
}

void calculateVelocityForces(float2 coords : TEXCOORD0,         // grid coordinates
                             out float4 result : COLOR,         // the resulting force
                             uniform float gravity,             // gravity parameter
                             uniform float buoyancy,            // buoyancy parameter
                             uniform float reactionThreshold,   // reaction threshold
                             uniform float vorticityWeight,     // vorticity weight
                             uniform samplerRECT densityField,  // density field
                             uniform samplerRECT oldForces)     // old force field
{
    result = f4texRECT(oldForces, coords);

    float4 density = f4texRECT(densityField, coords);

    float temperature = density.z;

    if (temperature < reactionThreshold)
        result.xz *= vorticityWeight;

    result.y += dot(float4(-gravity, 0, buoyancy, 0), density); // -gravity * exhaust + buoyancy * temperature
}

void calculateDivergenceForces(float2 coords : TEXCOORD0,           // grid coordinates
                               out float4 result : COLOR,           // the resulting forces
                               uniform float burningRate,           // burning rate parameter
                               uniform float stoichiometricMixture, // oxygen mixture rate
                               uniform float threshold,             // reaction threshold
                               uniform float massExpansion,         // mass expansion strength
                               uniform samplerRECT densityField)    // density field
{
    float4 density = f4texRECT(densityField, coords);

    float fuel = density.y;
    float temperature = density.z;

    result = float4(0,0,0,0);

    if (temperature > threshold) {

        float C = burningRate * stoichiometricMixture * fuel;

        C *= massExpansion;

        result = float4(C, C, C, C);
    }
}

void calculateColors(float2 coords : TEXCOORD0,        // grid coordinates
                     out float4 result : COLOR,        // the resulting color
                     uniform float scale,              // temperature scaling factor
                     uniform samplerRECT densityField, // density field
                     uniform samplerRECT colorTable)   // the blackbody radiation table
{
    float4 density = f4texRECT(densityField, coords);

    float exhaust = density.x;
    float temperature = density.z;

    result.rgb = f3texRECT(colorTable, float2(scale * temperature, 0.5));

    result.rgb *= exhaust;
    result.a = (result.x + result.y + result.z) / 3.0f;
}

void calculateColors2d(float2 coords : TEXCOORD0,        // grid coordinates
                        out float4 result : COLOR,        // the resulting color
                        uniform float scale,              // temperature scaling factor
                        uniform samplerRECT densityField, // density field
                        uniform samplerRECT colorTable)   // the blackbody radiation table
{
    float4 density = f4texRECT(densityField, coords);

    float exhaust = density.x;
    float temperature = density.z;

    result.rgb = f3texRECT(colorTable, float2(scale * temperature * exhaust, 0.5));

    result.a = (result.x + result.y + result.z) / 3.0f;
}

void calculateSmokeColors(float2 coords : TEXCOORD0,         // grid coordinates
                          out float4 result : COLOR,         // the resulting color
                          uniform float temperatureScale,    // temperature/fire color scale
                          uniform float smokeDensity,        // density of smoke
                          uniform float4 smokeColor,         // color of smoke
                          uniform float visibilityThreshold, // exhaust gas threshold
                          uniform samplerRECT fireDensities) // fire density fields
{
    float4 density = f4texRECT(fireDensities, coords);

    float exhaust = density.x;
    float temperature = density.z;

    if (temperature * temperatureScale > 3.14159)
        temperature = 0;
    else
        temperature = (cos(temperature*temperatureScale) + 1) / 2;

    float intensity = exhaust * temperature * smokeDensity;

    result = float4(smokeColor.xyz, intensity);

    if (exhaust < visibilityThreshold)
        result.a = 0;
}
```

## D.4 Particle system

```
#include "util.cg"

float4 lookupValue(uniform samplerRECT flat3dtexture,    // flat 3d texture
                   uniform samplerRECT offsetLookup,     // offset lookup table
                   uniform float3 position,              // value position
                   uniform float3 computationDimensions, // computation dimensions
                   uniform float scale)                  // domain scale
{
    float3 local = position.xyz / scale + computationDimensions * float3(0.5, 0, 0.5);

    return lookup3d(local, flat3dtexture, offsetLookup, computationDimensions);
}
```

```
void setVelocitiesSlice(float2 coords : TEXCOORD0,         // particle coordinates
                        out float4 result : COLOR,         // resulting velocity
                        uniform sampler2D positions,       // particle positions
                        uniform samplerRECT fluidVelocity, // velocity field
                        uniform samplerRECT sliceLookup,   // slice lookup table
                        uniform float computationWidth,    // computation width
                        uniform float computationHeight,   // computation height
                        uniform float sliceCount,          // slice count
                        uniform float scale,               // domain scale
                        uniform float voxelSize)           // voxel size
{
    float3 position = tex2D(positions, coords).xyz;

    result = lookupVelocitySlice(fluidVelocity, sliceLookup, position,
                                 float2(computationWidth,
                                        computationHeight),
                                 sliceCount,
                                 scale);

    result *= scale * voxelSize;
}


void setColorsSlice(float2 coords : TEXCOORD0,       // particle coordinates
                    out float4 result : COLOR,       // resulting color
                    uniform sampler2D positions,     // particle positions
                    uniform samplerRECT fluidColors, // velocity field
                    uniform samplerRECT sliceLookup, // slice lookup table
                    uniform float computationWidth,  // computation width
                    uniform float computationHeight, // computation height
                    uniform float sliceCount,        // slice count
                    uniform float scale)             // domain scale
{
    result = lookupSlice(fluidColors, sliceLookup,
                         tex2D(positions, coords).xyz,
                         float2(computationWidth,
                                computationHeight),
                         sliceCount,
                         scale);
}


void setVelocities(float2 coords : TEXCOORD0,         // particle coordinates
                   out float4 result : COLOR,         // resulting velocity
                   uniform sampler2D positions,       // particle positions
                   uniform samplerRECT fluidVelocity, // velocity field
                   uniform samplerRECT offsetLookup,  // offset lookup table
                   uniform float computationWidth,    // computation width
                   uniform float computationHeight,   // computation height
                   uniform float computationDepth,    // computation depth
                   uniform float scale,               // domain scale
                   uniform float voxelSize)           // voxel size
{
    float4 position = tex2D(positions, coords);

    result = lookupValue(fluidVelocity, offsetLookup, position,
                         float3(computationWidth,
                                computationHeight,
                                computationDepth),
                         scale);

    result *= scale * voxelSize;
}
```

```
void setColors(float2 coords : TEXCOORD0,        // particle coordinates
               out float4 result : COLOR,        // resulting color
               uniform sampler2D positions,      // particle positions
               uniform samplerRECT fluidColors,  // color field
               uniform samplerRECT offsetLookup, // offset lookup table
               uniform float computationWidth,   // computation width
               uniform float computationHeight,  // computation height
               uniform float computationDepth,   // computation depth
               uniform float scale)              // domain scale
{
    result = lookupValue(fluidColors, offsetLookup,
                         tex2D(positions, coords),
                         float3(computationWidth,
                                computationHeight,
                                computationDepth),
                         scale);
}


void updateVelocities(float2 coords : TEXCOORD0,         // particle coordinates
                      out float4 result : COLOR,         // resulting velocity
                      uniform float time,                // current time
                      uniform float timestep,            // current timestep
                      uniform float gravity,             // gravity force (optional)
                      uniform sampler2D positions,       // particle positions
                      uniform sampler2D spawnVelocities, // particle spawn velocities
                      uniform sampler2D velocities)      // particle velocities
{
    float4 position = tex2D(positions, coords);
    float spawnTime = position.w;

    float3 spawnVelocity = tex2D(spawnVelocities, coords).xyz;

    if (time > spawnTime) { // not sleeping
        result.xyz = tex2D(velocities, coords).xyz - float3(0, gravity * timestep, 0);

        if (position.y < 0)
            result.xyz = spawnVelocity;
    } else { // sleeping
        result.xyz = spawnVelocity;
    }
}


void advectParticles(float2 coords : TEXCOORD0,   // particle coordinates
                     out float4 result : COLOR,   // resulting position
                     uniform float4 distance,     // distance to advect
                     uniform sampler2D positions) // particle positions
{
    result = tex2D(positions, coords) + distance;
}


void updatePositions(float2 coords : TEXCOORD0,         // particle coordinates
                     out float4 result : COLOR,         // resulting position
                     uniform float time,                // current time
                     uniform float timestep,            // current timestep
                     uniform float gravity,             // gravity (not used)
                     uniform float visibilityThreshold, // alpha threshold for respawning
                     uniform float minimumLifeTime,     // minimum particle life time
                     uniform sampler2D particleColors,  // particle colors
                     uniform sampler2D positions,       // particle positions
                     uniform sampler2D spawnPositions,  // particle spawn positions
                     uniform sampler2D velocities)      // particle velocities
{
    float4 position = tex2D(positions, coords);
    float3 spawnPosition = tex2D(spawnPositions, coords).xyz;

    float spawnTime = position.w;

    if (time > spawnTime) { // not sleeping
        result.xyz = position.xyz + tex2D(velocities, coords).xyz * timestep;
        result.w = spawnTime;

        float4 color = tex2D(particleColors, coords);

        if (time > spawnTime + minimumLifeTime && color.a < visibilityThreshold) {
            result.w = time;
            result.xyz = spawnPosition;
        }
    } else { // sleeping
        result.xyz = spawnPosition;
        result.w = spawnTime;
    }
}


struct vertex_in // input to renderParticles
{
    float4 position : POSITION; // xy: particle index, zw: offset
    float2 coords : TEXCOORD0;  // texture coordinates
};


struct vertex_out // output from renderParticles
{
    float4 pos    : POSITION;   // position of particle quad vertex
    float2 coords : TEXCOORD0;  // texture coordinates
    float4 color  : COLOR0;     // particle color
};


vertex_out renderParticles(vertex_in vin,                          // input data
                           uniform float particleSize,             // particle size
                           uniform float time,                     // current time
                           uniform sampler2D positions : TEXUNIT1, // particle positions
                           uniform sampler2D colors : TEXUNIT2,    // particle colors
                           uniform float4x4 modelViewProjection)   // mvp matrix
{
    vertex_out vout;

    float4 pos = tex2D(positions, vin.position.xy);

    float3 right = normalize(modelViewProjection._m00_m01_m02) * vin.position.z;
    float3 up = normalize(modelViewProjection._m10_m11_m12) * vin.position.w;

    pos.xyz += (right + up) * particleSize;

    float spawnTime = pos.w;
    pos.w = 1;
    vout.pos = mul(modelViewProjection, pos);

    if (spawnTime > time)
        vout.pos.w = 0;

    vout.color = tex2D(colors, vin.position.xy);
    vout.coords = vin.coords;

    return vout;
}
```

## D.5 Volume renderer

```
#include "util.cg"

void displaySlice(float2 coords : TEXCOORD0,         // grid coordinates
                  out float4 result : COLOR,         // resulting color
                  uniform samplerRECT flat3dTexture, // flat 3d texture
                  uniform samplerRECT offsetLookup,  // offset lookup table
                  uniform float slice)               // slice index
{
    result = f4texRECTbilerp(flat3dTexture, f2texRECT(offsetLookup, slice) + coords);
}


void entryPointDetermination(float3 position : TEXCOORD0, // domain position
                             out float4 result: COLOR)    // domain position
{
    result = float4(position, 0);
}


void rayDirectionDetermination(float3 position : TEXCOORD0,     // domain position
                               float2 screen : WPOS,            // screen texture coordinates
                               uniform samplerRECT entryPoints, // ray entry points
                               out float4 result : COLOR)       // ray direction + length
{
    result.xyz = position - f3texRECT(entryPoints, screen);
    result.w = length(result.xyz);
    result.xyz /= result.w;
}


void rayMarcher(float3 position : TEXCOORD0,       // domain position
                float2 screen : WPOS,              // screen texture coordinates
                out float4 result : COLOR,         // resulting color
                uniform samplerRECT flat3dTexture, // volume texture
                uniform samplerRECT flat3dOffset,  // offset lookup table
                uniform samplerRECT rayDirections, // ray directions
                uniform float domainWidth,         // domain width
                uniform float domainHeight,        // domain height
                uniform float domainDepth,         // domain depth
                uniform float stepSize)            // ray marching step size
{
    float t = stepSize;
    float4 accumulatedColor = 0;
    float4 ray = texRECT(rayDirections, screen);

    while (t < ray.w) {
        float3 samplePos = position + t * ray.xyz;

        float4 sampleColor = lookup3d(samplePos, flat3dTexture, flat3dOffset,
                                      float3(domainWidth, domainHeight, domainDepth));

        accumulatedColor = sampleColor + accumulatedColor * (1 - sampleColor);
        t += stepSize;
    }

    result = accumulatedColor * stepSize;
}


void rayMarcherSlice(float3 position : TEXCOORD0,        // domain position
                     float2 screen : WPOS,               // screen texture coordinates
                     out float4 result : COLOR,          // resulting color
                     uniform samplerRECT flat3dTexture,  // volume texture
                     uniform samplerRECT sliceLookup,    // slice lookup texture
                     uniform samplerRECT rayDirections,  // ray directions
                     uniform float domainWidth,          // domain width
                     uniform float domainHeight,         // domain height
                     uniform float domainDepth,          // domain depth
                     uniform float stepSize)             // ray marching step size
{
    float t = stepSize;
    float4 accumulatedColor = 0;
    float4 ray = texRECT(rayDirections, screen);
    float3 offset = float3(domainWidth, 0, domainWidth) * 0.5;

    while (t < ray.w) {
        float3 samplePos = position + t * ray.xyz;

        float4 sampleColor = lookupSlice(flat3dTexture,
                                         sliceLookup,
                                         samplePos - offset,
                                         float2(domainWidth, domainHeight),
                                         domainDepth,
                                         1);

        accumulatedColor = sampleColor + accumulatedColor * (1 - sampleColor);

        t += stepSize;
    }

    result = accumulatedColor;
}
```

## D.6 Dynamic illumination

```
#include "util.cg"

struct VertexInput
{
    float4 position : POSITION;  // untransformed vertex position
    float3 normal : NORMAL;      // untransformed normal
    float2 texCoord : TEXCOORD;  // texture coordinates
    float4 color : COLOR;        // material color
};


struct VertexOutput
{
    float4 position : POSITION;        // transformed position
    float2 texCoord : TEXCOORD0;       // texture coordinates
    float4 color : COLOR;              // material color
    float3 objectPosition : TEXCOORD1; // untransformed position
    float3 objectNormal : TEXCOORD2;   // untransformed normal
```

```
};


VertexOutput fragmentLightingVS(VertexInput vin,                      // input data
                                uniform float4x4 modelViewProjection) // mvp matrix
{
    VertexOutput vout;

    vout.position = mul(modelViewProjection, vin.position);

    vout.objectPosition = vin.position.xyz;
    vout.objectNormal = vin.normal;
    vout.color = vin.color;
    vout.texCoord = vin.texCoord;

    return vout;
}


struct FragmentInput // interpolated vertex data
{
    float2 texCoord : TEXCOORD0;       // texture coordinates
    float4 color : COLOR;              // material color
    float3 objectPosition : TEXCOORD1; // untransformed position
    float3 objectNormal : TEXCOORD2;   // untransformed normal
};


float3 lighting(float3 P,             // surface position
                float3 N,             // surface normal
                float3 lightPosition, // light position
                float3 lightColor,    // light color
                float3 eyePosition,   // eye position
                float kd,             // diffuse coefficient
                float ks,             // specular coefficient
                float shininess)      // shininess
{
    // diffuse
    float3 L = normalize(lightPosition - P);
    float distance = length(lightPosition - P);
    float attenuation = 1 / (distance * distance);
    float diffuseLight = max(dot(N, L), 0);
    float3 diffuse = kd * lightColor * attenuation * diffuseLight;

    // specular
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0), shininess);
    if (diffuseLight <= 0) specularLight = 0;
    float3 specular = ks * lightColor * attenuation * specularLight;

    return diffuse + specular;
}


void fragmentLightingTexturedFS(FragmentInput fin,                 // input data
                                out float4 color : COLOR,          // resulting color
                                uniform sampler2D texture,         // material texture
                                uniform float3 eyePosition,        // eye position in object space
                                uniform samplerRECT lightPositions, // light positions in object space
                                uniform samplerRECT lightColors,   // light colors
                                uniform int numLights,             // light count
                                uniform float3 globalAmbient,      // global ambient color
                                uniform float ka,                  // ambient coefficient
                                uniform float kd,                    // diffuse coefficient
                                uniform float ks,                    // specular coefficient
                                uniform float shininess)             // shininess
{
    float3 P = fin.objectPosition;
    float3 N = normalize(fin.objectNormal);

    float3 accumulatedLight = ka * globalAmbient;

    for (int i = 0; i < numLights; ++i) {
        float3 lightPosition = f3texRECT(lightPositions, float2(i + 0.5, 0.5));
        float3 lightColor = f3texRECT(lightColors, float2(i + 0.5, 0.5));

        accumulatedLight += lighting(P, N, lightPosition, lightColor, eyePosition, kd, ks, shininess);
    }

    color.xyz = tex2D(texture, fin.texCoord).xyz * accumulatedLight;
    color.w = 1;
}


void fragmentLightingFS(FragmentInput fin,                  // input data
                        out float4 color : COLOR,           // resulting color
                        uniform float3 eyePosition,         // eye position in object space
                        uniform samplerRECT lightPositions, // light positions in object space
                        uniform samplerRECT lightColors,    // light colors
                        uniform int numLights,              // light count
                        uniform float3 globalAmbient,       // global ambient color
                        uniform float ka,                   // ambient coefficient
                        uniform float kd,                   // diffuse coefficient
                        uniform float ks,                   // specular coefficient
                        uniform float shininess)            // shininess
{
    float3 P = fin.objectPosition;
    float3 N = normalize(fin.objectNormal);

    float3 accumulatedLight = ka * globalAmbient;

    for (int i = 0; i < numLights; ++i) {
        float3 lightPosition = f3texRECT(lightPositions, float2(i + 0.5, 0.5));
        float3 lightColor = f3texRECT(lightColors, float2(i + 0.5, 0.5));
        accumulatedLight +=
            lighting(P, N, lightPosition, lightColor, eyePosition, kd, ks, shininess);
    }

    color.xyz = fin.color.xyz * accumulatedLight;
    color.w = 1;
}


void sampleLightColors(float2 coords : TEXCOORD0,          // texture coordinates
                       out float4 result : COLOR,          // resulting light color
                       uniform samplerRECT fireColorField, // fire color field
                       uniform samplerRECT lightPositions, // light positions
                       uniform samplerRECT lightColors,    // base light colors
                       uniform samplerRECT offsetLookup,   // offset lookup table
                       uniform float computationWidth)     // width of computational domain
{
    float3 lightPosition = f3texRECT(lightPositions, coords) + float3(0.5, 0, 0.5);
    lightPosition *= computationWidth;

    float2 localCoords = f2texRECT(offsetLookup, lightPosition.z);

    result = texRECT(fireColorField, localCoords + lightPosition.xy);
    result = (result.x + result.y + result.z) / 3;
    result *= texRECT(lightColors, coords);
}


void sampleLightColorsSlice(float2 coords : TEXCOORD0,          // texture coordinates
                            out float4 result : COLOR,          // resulting light color
                            uniform samplerRECT fireColorField, // fire color field
                            uniform samplerRECT lightPositions, // light positions
                            uniform samplerRECT lightColors,    // base light colors
                            uniform samplerRECT sliceLookup,    // slice lookup table
                            uniform float computationWidth,     // width of computational domain
                            uniform float computationHeight,    // height of computational domain
                            uniform float sliceCount)           // number of slices
{
    float3 lightPosition = f3texRECT(lightPositions, coords);

    float offset = computationWidth / 2;

    lightPosition = float3(lightPosition.x, lightPosition.y, lightPosition.z);

    result = lookupSlice(fireColorField,
                         sliceLookup,
                         lightPosition,
                         float2(computationWidth, computationHeight),
                         sliceCount,
                         1 / computationWidth);

    result = (result.x + result.y + result.z) / 3;
    result *= texRECT(lightColors, coords);
}
```

## D.7 Utility functions

```
// util.cg

float4 f4texRECTbilerp(samplerRECT tex, // input texture
                       float2 s)        // sampling position
{
    float4 st;
    st.xy = floor(s - 0.5) + 0.5;
    st.zw = st.xy + 1;

    float2 t = s - st.xy; //interpolating factors

    float4 tex11 = f4texRECT(tex, st.xy);
    float4 tex21 = f4texRECT(tex, st.zy);
    float4 tex12 = f4texRECT(tex, st.xw);
    float4 tex22 = f4texRECT(tex, st.zw);

    // bilinear interpolation
    return lerp(lerp(tex11, tex21, t.x), lerp(tex12, tex22, t.x), t.y);
}


float4 f4tex2Dbilerp(sampler2D tex,  // input texture
                     float2 s,       // sampling position
                     float2 dim)     // texture dimensions
{
    s *= dim;
    float4 st;
    st.xy = floor(s - 0.5) + 0.5;
    st.zw = st.xy + 1;

    float2 t = s - st.xy; //interpolating factors

    st /= float4(dim, dim);

    float4 tex11 = tex2D(tex, st.xy);
    float4 tex21 = tex2D(tex, st.zy);
    float4 tex12 = tex2D(tex, st.xw);
    float4 tex22 = tex2D(tex, st.zw);

    // bilinear interpolation
    return lerp(lerp(tex11, tex21, t.x), lerp(tex12, tex22, t.x), t.y);
}


float3 f3clamp(float3 value, // value to clamp
               float3 min,   // minimum value
               float3 max)   // maximum value
{
    value = (value - min) / (max - min); // 1 / (max - min) could be precalculated

    saturate(value);

    return min + value * (max - min);
}


float4 lookup3d(float3 localCoords,       // local grid coordinates
                samplerRECT value,        // quantity to interpolate
                samplerRECT offsetLookup, // offset lookup table
                float3 dimensions)        // domain dimensions
{
    // calculate low and high interpolation coordinates
    float3 low = floor(localCoords - 0.5) + 0.5;
    float3 high = low + 1;

    // clamp coordinates to domain
    low = f3clamp(low, 0.5, dimensions - 0.5);
    high = f3clamp(high, 0.5, dimensions - 0.5);

    float4 st, temp;

    st.xy = low.xy;
    st.zw = high.xy;
    temp.z = low.z;
    temp.w = high.z;

    // lookup offsets
    float2 lowOffset = f2texRECT(offsetLookup, temp.z);
    float2 highOffset = f2texRECT(offsetLookup, temp.w);

    // locate all 8 neighbours
    float4 tex111 = f4texRECT(value, lowOffset + st.xy);
    float4 tex211 = f4texRECT(value, lowOffset + st.zy);
    float4 tex121 = f4texRECT(value, lowOffset + st.xw);
    float4 tex221 = f4texRECT(value, lowOffset + st.zw);
    float4 tex112 = f4texRECT(value, highOffset + st.xy);
    float4 tex212 = f4texRECT(value, highOffset + st.zy);
    float4 tex122 = f4texRECT(value, highOffset + st.xw);
    float4 tex222 = f4texRECT(value, highOffset + st.zw);
    float2 t = localCoords.xy - st.xy; // xy-interpolation factors

    float u = localCoords.z - temp.z; // z-interpolating factor

    // interpolate in each depth plane
    float4 lowLerp = lerp(lerp(tex111, tex211, t.x), lerp(tex121, tex221, t.x), t.y);
    float4 highLerp = lerp(lerp(tex112, tex212, t.x), lerp(tex122, tex222, t.x), t.y);

    // interpolate between depth plane
    return lerp(lowLerp, highLerp, u);
}


float4 lookupSlice(uniform samplerRECT flat3dtexture,    // flat 3d texture
                   uniform samplerRECT sliceLookup,      // slice lookup table
                   uniform float3 position,              // position to sample
                   uniform float2 computationDimensions, // width and height of domain
                   uniform float sliceCount,             // slice count
                   uniform float scale)                  // domain scale
{
    float3 local = position.xyz / scale; // + computationDimensions.xyx * float3(0.5, 0, 0.5);

    float radius = length(local.xz);

    float4 result;

    if (2 * radius > (computationDimensions.x - 1) ||
                     local.y < 0.5 ||
                     local.y > (computationDimensions.y - 0.5))
    {
        result = float4(0, 0, 0, 0);
    }
    else
    {
        float lookupIndex = ((atan2(local.z, local.x) / 3.14159) + 1.0) * sliceCount + 0.5;

        float low = floor(lookupIndex - 0.5) + 0.5;
        float high = low + 1.0;

        float t = lookupIndex - low; // interpolating factor

        float3 previous = f3texRECT(sliceLookup, float2(low, 0.5));
        float3 next = f3texRECT(sliceLookup, float2(high, 0.5));

        float2 previousOffset = previous.xy + float2(radius * previous.z, local.y);
        float2 nextOffset = next.xy + float2(radius * next.z, local.y);

        float4 previousColor = f4texRECTbilerp(flat3dtexture, previousOffset);
        float4 nextColor = f4texRECTbilerp(flat3dtexture, nextOffset);

        result = lerp(previousColor,
                      nextColor,
                      t);
    }

    return result;
}


float4 lookupVelocitySlice(uniform samplerRECT flat3dtexture,    // flat 3d texture
                           uniform samplerRECT sliceLookup,      // slice lookup table
                           uniform float3 position,              // sample position
                           uniform float2 computationDimensions, // width and height of domain
                           uniform float sliceCount,             // slice count
                           uniform float scale)                  // domain scale
{
    float3 local = position.xyz / scale; // + computationDimensions.xyx * float3(0.5, 0, 0.5);

    float radius = length(local.xz);

    float4 result;

    if (2 * radius > (computationDimensions.x - 1) ||
                      local.y < 0.5 ||
                      local.y > (computationDimensions.y - 0.5))
    {
        result = float4(0, 0, 0, 0);
    }
    else
    {
        float lookupIndex = ((atan2(local.z, local.x) / 3.14159) + 1.0) * sliceCount + 0.5;

        float low = floor(lookupIndex - 0.5) + 0.5;
        float high = low + 1.0;

        float t = lookupIndex - low; // interpolating factor

        float3 previous = f3texRECT(sliceLookup, float2(low, 0.5));
        float3 next = f3texRECT(sliceLookup, float2(high, 0.5));

        float2 previousOffset = previous.xy + float2(radius * previous.z, local.y);
        float2 nextOffset = next.xy + float2(radius * next.z, local.y);

        float2 previousColor =
            f4texRECTbilerp(flat3dtexture, previousOffset).xy * float2(previous.z, 1);
        float2 nextColor =
            f4texRECTbilerp(flat3dtexture, nextOffset).xy * float2(next.z, 1);

        float2 interpolated = lerp(previousColor,
                                   nextColor,
                                   t);
        result.y = interpolated.y;
        result.xz = interpolated.x * local.xz / radius;
    }

    return result;
}
```

EG UK Theory and Practice of Computer Graphics (2006)
M. McDerby, L. Lever (Editors)

# Physically Based Simulation and Visualization of Fire in Real-Time using the GPU

Samuel Rødal[†], Geir Storli[‡], and Odd Erik Gundersen[§]

Visualization Group, Department of Computer and Information Science, NTNU, Norway

**Abstract**

*In this paper we present a physically based framework for real-time simulation and visualization of fire using the GPU. The physics of fire is modeled through a combination of a fluid solver and a combustion process causing the characteristic motion of fire. The simulation results are then rendered using a particle system combined with a black-body radiation model where the physically based simulation governs both the motion and appearance of the particles. By performing individual slice simulations in 2D and combining them using volumetric extrusion we achieve better performance than by performing the simulation in 3D without compromising the visual quality. Thus, achieving our goal of visualizing bonfire and torch-like fire effects with high visual quality in real-time.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

## 1. Introduction

Fire is a powerful effect and can be used in virtual environments like games in order to increase immersion, suspending the disbelief of the user. However, it is hard to reproduce the chaotic and turbulent behavior of fire using traditional procedural methods. Fire is a very intense visual phenomenon, and thus needs to be realistically reproduced in order to convince the eye of a human observer.

Our view is that the laws of nature play an important part and must be taken into consideration when simulating visually convincing fire. Thus, we take a physically based approach to the simulation and visualization of fire. This also reduces the need to come up with various ad hoc approximations which can be hard to justify. With the increased power and programmability of modern GPUs, more processing power is available for performing physically based simulations while still achieving real-time frame rates.

Our main contributions are performing the combustion process and fluid simulation from [MK02] on the GPU and using a black-body radiation model in combination with a particle system also running on the GPU. The combustion simulation is combined with vorticity confinement as used in [NFJ02, WLL04, KW05]. The black-body radiation color table is precomputed and stored in a lookup texture on the GPU and used when calculating the color radiated by the hot exhaust gas. Another contribution is combining the combustion process with volumetric extrusion [RNGF03, KW05]. We have achieved our goal of visualizing bonfire and torch-like fire effects with high visual quality in real-time.

This paper is organized as follows. After a brief overview of related work in chapter 2, we describe both the theory and implementation of our framework for simulating and visualizing fire in chapter 3. In chapter 4 we present the results and analysis and chapter 5 concludes with a summary and future work.

## 2. Related work

We divide between methods used for simulating and methods used for visualizing fire and will treat them separately. Although other techniques have achieved visually pleasing

[†] knuterro@idi.ntnu.no
[‡] geirst@idi.ntnu.no
[§] odderik@idi.ntnu.no

results [Ngu04, LF02], our focus is on physically based simulation techniques.

### 2.1. Simulating fire

[NFJ02] present an offline physically based method for modeling fire. A thin flame model is developed, using an implicit surface to represent the reaction zone where vaporized fuel reacts with oxygen and creates hot gaseous products. The movement of the implicit surface is tracked using the level set method, and the flow of vaporized fuel and the flow of hot gaseous products are modeled independently using the incompressible Navier-Stokes equations.

[MK02] model fire in real-time by using a combustion process in addition to the incompressible Navier-Stokes equations. The equations are discretized into a 3D grid and solved using a fluid solver similar to the approach in [Sta99]. The fluid solver is used to control the motion of a three-gas system consisting of oxidizing air, fuel gases, and exhaust gases. The combustion process models the reaction that occurs between oxygen and fuel gases at a certain temperature threshold, resulting in the generation of exhaust gases and the spread of heat.

[WLMK02] use the Lattice Boltzmann model instead of the Navier-Stokes equations to simulate the fire process using a temperature field to model the generation of smoke. The fire behavior is mainly affected by the direction of wind and the location of fuel and non-burning objects.

[KW05] use volumetric extrusion of a 2D fluid simulation to simulate fire. However, instead of using a physically based combustion model to guide the fluid simulation they use pressure templates to disturb the flow in order to create a turbulent fire. Heat is modeled and used to scale the pressure templates to create more turbulence at the base of the fire.

The simulation is performed on the GPU in [WLMK02] and [KW05], and on the CPU in [NFJ02] and [MK02].

### 2.2. Visualizing fire

In [NFJ02] a Monte Carlo ray tracing approach is used to visualize the fire, treating the hot gaseous products as a participating medium. The radiance emitted by the medium is modeled using black-body radiation.

A hardware based volume rendering technique is used in [MK02] for fire visualization. The output of the fire simulation is voxelized data of the fuel gas, exhaust gas, and heat in the system, and each voxel is replaced by a semitransparent polygon where the level of transparency is controlled by the density of fuel gas and exhaust gas in the voxel. The fuel gas is shown in yellow and the reaction zone where the combustion occurs is shown in red.

Texture splats are used to visualize the fire in [WLMK02]. The velocity volume from the simulation is used to advect the display primitives, which are removed from the system when the fuel they are holding is consumed by combustion. The display primitives are rendered using texture splatting.

[KW05] use a GPU implemented particle system to visualize the fire. The velocity vector field from the fluid simulation is used to update the particle positions, and the particles are rendered using textured point sprites.

Recently, the ability of the GPU to perform both the simulation and visualization of particle systems has been explored. [Lat04] and [KSW04] introduce a full GPU implementation of the simulation and rendering of a particle system. The particle positions are stored in a 2D texture on the GPU and the current particle velocities in another 2D texture. The velocity texture and position texture are both updated in separate rendering passes. In [KSW04] the updated particle positions are rendered into a vertex array memory object. This array is processed when rendering the particles to the screen. [Lat04] propose the use of vertex texture fetch to read the particle positions from a vertex shader responsible for rendering the particles to the screen.

## 3. Simulating and visualizing fire on the GPU

Here we first give a general overview and then separately present our approach for simulating fire and our approach for visualizing fire. Then, we present the complete algorithm, and finally the implementation details are discussed.

### 3.1. Overview

We simulate the evolution of a fuel gas field, an exhaust gas field, and a temperature field, in co-evolution with a velocity field. The combustion process 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, which in combination with vorticity confinement causes the characteristic fire-like motion. All the simulation steps are efficiently performed on the GPU by packing the fuel gas, exhaust gas, and temperature field into a single texture. This packing is similarly performed for the components of the velocity field. We implement two variations of the simulation. One performs the simulation in full 3D and the other performs individual 2D slice simulations and combines them through volumetric extrusion.

We visualize the result of the fire simulation using a black-body radiation model and a particle system. A black-body radiation table is precomputed and stored as a lookup texture on the GPU. The black-body radiation approximates the radiation from the hot exhaust gas. Particle data is stored in textures on the GPU, which are updated by the particle system simulation and used by the vertex shader to specify the particle positions. The velocity field from the physically based simulation is used to advect the particles, while the temperature and exhaust gas fields are used in combination with the black-body radiation table to compute the particle colors.

### 3.2. Simulating fire

Our fire simulation is largely based on the approach presented in [MK02], which combines the stable fluid solver from [Sta99] with a combustion process modeling fuel gas, exhaust gas, and temperature fields. The fields are limited to the finite computational domain, which represents the area where the fire is located, and are discretized into a voxel or grid structure. In addition, we use the vorticity confinement method used in [NFJ02], [WLL04] and [KW05] to create a more turbulent flame.

#### 3.2.1. Velocity field

The velocity field $\mathbf{u}$ is a vector field specifying the direction and speed of air and gas. It is governed by the Navier-Stokes equations for incompressible flow with zero viscosity, shown in equation 1 and equation 2.

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

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

The first term on the right-hand side of equation 1 is the self-advection of the velocity, causing velocity to be pushed along itself. The second term, $-\nabla p$, is the gradient of the pressure field, causing velocity to move from areas of high pressure to areas of low pressure. The pressure field is used as a correcting term, ensuring that equation 2 holds by projecting the velocity field onto its divergence free component. The velocity field needs to be divergence free in order to conserve mass according to equation 2.

The last term on the right hand side of equation 1 is the external force acting on the velocity field. The external force consists of several separate forces, shown in equation 3:

$$\mathbf{F} = f_{vorticity} + f_{gravity} + f_{buoyancy}, \tag{3}$$

where $f_{vorticity}$ is the vorticity confinement force, $f_{gravity}$ is the gravity force due to fuel and exhaust gases, and $f_{buoyancy}$ is the buoyancy force due to heat. These forces are described later in equations 9, 10, and 11.

#### 3.2.2. Fire density fields

We use the fire density fields as a common term for the scalar fuel gas, exhaust gas, and temperature fields. To create a realistic and moving flame, we use the stable fluid solver and the velocity field to advect the fire density fields throughout the computational domain.

We use a slightly modified version of the density advection equation presented in [Sta99] to describe the evolution of the three fire density fields: fuel gas $g$, exhaust gas $a$, and temperature $T$. The modified equations governing the evolution of respectively the fuel gas field, exhaust gas field, and temperature field are given by 4, 5, and 6. These three equations correspond to the equations used in [MK02].

$$\frac{\partial g}{\partial t} = -\mathbf{u}\cdot\nabla g + \kappa_g \nabla^2 g - \alpha_g g + S_g + C_g \tag{4}$$

$$\frac{\partial a}{\partial t} = -\mathbf{u}\cdot\nabla a + \kappa_a \nabla^2 a - \alpha_a a + C_a \tag{5}$$

$$\frac{\partial T}{\partial t} = -\mathbf{u}\cdot\nabla T + \kappa_T \nabla^2 T - \alpha_T T + S_T + C_T \tag{6}$$

The first term on the right hand side of each equation is the advection term, causing the fire density fields to be carried along by the velocity field. The second term is the diffusion term, simulating the tendency of the densities to spread out. The third term governs the dissipation of the fire density fields, and is controlled by the dissipation rates $\alpha_g$, $\alpha_a$, and $\alpha_T$. We also have source terms, $S_g$ for fuel gas and $S_T$ for temperature, which are used to inject fuel gas and temperature in order to get the fire started and to keep it going. The remaining terms $C_g$, $C_a$, and $C_T$ are related to combustion, and are discussed in detail in section 3.2.5.

#### 3.2.3. Vorticity confinement

To achieve real-time results, we have to use a rather coarse grid in our simulation. In addition to this, the stable fluid solver suffers from some numerical dissipation, meaning that much of the low-level turbulence that is important for achieving a realistic flame is lost. To counterbalance this, we use the vorticity confinement method, also used by [NFJ02], [WLL04], and [KW05], to find the vortices that are formed in the velocity field. A vortex is the center of a rotational movement. The vorticity at a given field point thus measures how much rotational movement is present there. The vorticity $\omega$ of the velocity field $\mathbf{u}$ is calculated with equation 7.

$$\omega = \nabla \times \mathbf{u} \tag{7}$$

Next, the normalized gradient $\mathbf{N}$ of $|\omega|$, pointing from lower to higher concentrations of vorticity, is calculated from equation 8, and finally we calculate the vorticity force $f_{vorticity}$, given in equation 9. The parameter $\varepsilon$ controls the strength of the vorticity confinement, and $h$ is the distance between two grid cells.

$$\mathbf{N} = \frac{\nabla|\omega|}{|\nabla|\omega||} \tag{8}$$

$$f_{vorticity} = \varepsilon h\,(\mathbf{N} \times \omega) \tag{9}$$

### 3.2.4. Gravity and buoyancy

Fuel gas and exhaust gas are pulled down due to a gravity force acting on the velocity field. The gravity force is given by the following equation:

$$f_{gravity} = f_g\,(g+a)\begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{10}$$

The constant $f_g$ determines the strength of the gravity force, while $g$ and $a$ are the amount of fuel gas and exhaust gas respectively.

Like the gravity force, the buoyancy force acts on the velocity field. The buoyancy force causes hot air to rise, and is given by the following equation:

$$f_{buoyancy} = f_b\,(T - T_{ambient})\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{11}$$

The strength of the buoyancy force is determined by the buoyancy constant $f_b$, the temperature $T$, and the ambient temperature constant $T_{ambient}$, which is the temperature of the surrounding air. To create realistic fire the buoyancy force is crucial, as rising air is one of the main causes of the characteristic and turbulent appearance of the flame.

### 3.2.5. Fire combustion

An important part of the physically based simulation is to take into account what happens when fuel gas reacts with oxygen creating exhaust gas and heat. To simulate the combustion we choose an approach similar to the one in [MK02].

The combustion will only occur if the temperature is above a given threshold temperature $T_{threshold}$. In contrast to [MK02] we assume that there will always be enough oxygen to react with the fuel gas, which simplifies the equation for the combustion parameter:

$$C = rbg, \tag{12}$$

where $r$ is the burning rate parameter describing how fast the fuel gas can be burned, $b$ is the stoichiometric mixture describing the amount of oxygen required to burn one unit of fuel, and $g$ is the amount of fuel gas. Equations 13, 14, and 15 use the combustion parameter $C$, and describe the rate of change of respectively the fuel gas, exhaust gas, and temperature due to combustion.

$$C_g = \begin{cases} -\frac{C}{b} & \text{if } T > T_{threshold} \\ 0 & \text{if } T \leq T_{threshold} \end{cases} \tag{13}$$

$$C_a = \begin{cases} C(1+\frac{1}{b}) & \text{if } T > T_{threshold} \\ 0 & \text{if } T \leq T_{threshold} \end{cases} \tag{14}$$

$$C_T = \begin{cases} T_0 C & \text{if } T > T_{threshold} \\ 0 & \text{if } T \leq T_{threshold} \end{cases} \tag{15}$$

Because we assume there will always be enough oxygen to react with the fuel gas, the parameter $b$ only controls how much oxygen is involved in the reaction, and will not directly affect how fast the fuel is consumed. In addition the parameter $T_0$ is used in equation 15 to control the amount of heat that is produced by the reaction.

## 3.3. Visualizing fire

Using a precomputed black-body radiation lookup table, a fire color field is computed based on the exhaust gas and temperature fields. Then, the fire is visualized using a particle system. The particle positions are updated based on the velocity field and the particle colors are read from the fire color field.

### 3.3.1. Computing the fire color field

We use Planck's formula for black-body radiation given by equation 16 in order to calculate the intensity radiated by the hot exhaust gas.

$$B_\lambda(T) = \frac{2\pi hc^2}{\lambda^5\left(e^{\frac{hc}{\lambda kT}} - 1\right)} \tag{16}$$

By using the wavelengths of red, green, and blue light and the temperature of the gas, we calculate the three intensities $B_{red}$, $B_{green}$, and $B_{blue}$. These intensities have a very high dynamic range whereas the resulting color should have a limited dynamic range suitable for display on traditional computer monitors. To map the given intensities onto a limited dynamic range, we use the exponential mapping function from [Mat97], shown in the following equation:

$$n = 1 - e^{\frac{-L}{L_{average}}}, \tag{17}$$

where $L$ is the original intensity, and $L_{average}$ is a constant controlling the overall brightness. The resulting intensity $n$ will be in the range $[0, 1\rangle$.

Equations 16 and 17 are used to precompute black-body radiation color values for a user specified range of temperatures, which are stored in a one dimensional lookup table.

At the beginning of each visualization step, the exhaust gas and temperature fields are used in combination with the black-body radiation lookup table in order to compute the fire color field. This is done for each cell in the computational domain. Equation 18 shows how the color **c** in the fire color field is computed, based on the temperature $T$, exhaust gas $a$, and a temperature scaling factor $T_{scale}$, which is used to control the resulting brightness of the fire. *lookup* is the black-body radiation lookup table.

$$\mathbf{c} = a \times lookup\,(T_{scale}T) \tag{18}$$

### 3.3.2. Particle system

Like [KW05], we visualize the fire using a particle system defined in the computational domain. Each particle represents a small element of the fire and has a set of associated variables: spawn position, current position, initial spawn delay, current velocity, and color. Spawn position and initial spawn time are given at the beginning of the simulation, whereas the other variables are dynamically updated. A particle's color is specified by an RGBA color value.

Initially, after the given spawn delay, a particle's position is set to the spawn position of the particle. A simple Euler step is later used to update a particle's position as described by the following equation:

$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{v}_i \delta t, \tag{19}$$

where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle $i$ respectively and $\delta t$ is the timestep. Based on the particle position, the particle's velocity and color are found by interpolating samples from the discretized simulation velocity and fire color fields.

When a particle's intensity drops below a certain threshold, it is respawned by resetting its position to its spawn position. A minimum initial lifetime ensures that the particle is not respawned before it has had a chance to enter the fire. Particles are textured like in [WLMK02], creating more low-level detail. Figure 1 shows an example of a texture we used for this.

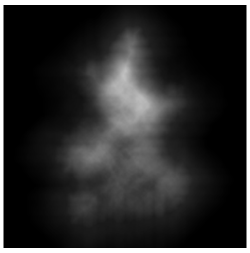

**Figure 1:** *Texture used for particles*

### 3.4. The complete algorithm

Both the fire simulation and the visualization are performed for each frame. The fire simulation evolves the velocity and fire density fields based on the combustion process and a stable fluid solver. The velocity and fire density fields are discretized in a grid structure throughout the computational domain and each step of the simulation is performed for all cells in the grid. The fire simulation steps are shown below:

1. Compute velocity forces and add them to the velocity field.
   - Vorticity confinement using equation 9.
   - Gravity using equation 10.
   - Buoyancy using equation 11.
2. Compute fire density forces and add them to the fuel gas, exhaust gas, and temperature fields.
   - Dissipation, part of equations 4, 5, and 6.
   - Source terms, part of equations 4 and 6.
   - Combustion forces using equations 13, 14, and 15.
3. Self-advect and project the velocity field based on equations 1 and 2 using the stable fluid solver.
4. Advect and diffuse the fuel gas, exhaust gas, and temperature fields based on equations 4, 5, and 6 using the stable fluid solver.

After the velocity and fire density fields have been calculated, they in turn are used by the particle system to visualize the fire. A discretized fire color field is now computed based on the temperature and exhaust gas fields, using the precomputed black-body radiation color table. The discretized velocity and fire color fields are then used by the particle system to locate corresponding velocity and color values based on the particle positions. This is shown below:

1. Compute the fire color field using the precomputed black-body radiation color table as described in equation 18.
2. Based on particle positions, sample particle velocities and colors from velocity and fire color fields respectively.
3. Advect particles based on the sampled velocities from step 2 using equation 19.
4. Render particles using a texture splat colored by the sampled colors from step 2.

### 3.5. Implementation details

To implement the fire simulation and particle simulation on the GPU, we use the FBO (framebuffer object) extension, which allows us to render directly to textures. Textures are used to store both the main velocity and fire density fields as well as particle data (including positions, velocities, and colors). Computations on textures are performed using Cg fragment shaders. GPU implementations of stable fluid solvers are presented in [LLW04] and [Har04].

We implement two versions of the algorithm. One performs a full 3D simulation and the other performs individual

2D slice simulations and combines them using volumetric extrusion [RNGF03, KW05]. In both cases we use flat 3D textures [HBSL03] to be able to simulate several slices at once (depth slices for the full 3D simulation and individual 2D slices for the slice simulation). Figure 2 shows an example of a flat 3D texture used to store two 2D slices of the fire color field.

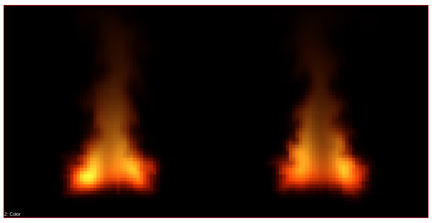

**Figure 2:** *A flat 3D texture representing two slices of the fire color field at grid size 64x64.*

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 a set of individual 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 [RNGF03] is used between the two closest slices and then bilinear interpolation is used within the two slices.

Particles are rendered as quads, which requires four vertices for each particle. As the particle positions are stored in textures on the GPU, we create a vertex buffer object. Each vertex contains the particle's index in the particle textures as well as an offset specifying the corner of the quad. The vertex shader then uses the index to read the position and color from the particle position and color textures respectively. The offset is transformed based on the modelview transformation matrix to make sure that the plane of the quad is parallel to the the viewing plane. Reading from textures in vertex shaders requires a GPU with VS3.0 support. Using a vertex buffer in combination with texture fetch in the vertex shader for particle system rendering was suggested by [Lat04].

## 4. Results and analysis

In this section, the performance of our algorithm is evaluated and it is compared to other approaches for visualizing fire. The limitations of the algorithm are also discussed.

### 4.1. Performance evaluation

In this section we first present and compare the performance results from full 3D and 2D slice simulations. We then use the 2D slice simulation coupled with the particle system and compare frame rates using different particle counts with and without rendering the particle system. We also present some visual results. All the tests were run on a 3 GHz Intel Pentium 4 CPU with 512 MB RAM and a NVIDIA GeForce 7800 GT with 256 MB VRAM.

Table 1 shows frame rates from the full 3D simulation with and without visualization. As expected, the frame rate rapidly decreases as we increase the grid dimensions. Also, the visualization is the most time-consuming part at lower grid sizes. As can be seen in figure 3, 32x32x32 is sufficient for achieving visually pleasing results in combination with the particle system.

| **Grid size** | **Fire simulation** | **Visualization** |
|---|---|---|
| 16x16x16 | 503.3 fps | 49.4 fps |
| 24x24x24 | 202.0 fps | 42.5 fps |
| 32x32x32 | 93.9 fps | 34.0 fps |
| 48x48x48 | 28.2 fps | 19.0 fps |

**Table 1:** *Frame rates for full 3D fire simulation, both with and without a particle system with 2048 particles.*

The frame rates achieved by running the 2D slice simulation with and without visualization using both 2 slices and 4 slices are presented in table 2. The results show that a large speed increase is gained from not performing the simulation in full 3D, although the difference is less explicit when visualizing. A simulation grid of 64x64 with two slices is sufficient for good visual results (figure 3). The frame rate is around 5 times as high as for the 3D simulation without visualization at 32x32x32.

|  | Fire simulation |  | Visualization |  |
|---|---|---|---|---|
| **Grid size** | **2 slices** | **4 slices** | **2 slices** | **4 slices** |
| 32x32 | 608.8 fps | 597.1 fps | 50.4 fps | 49.5 fps |
| 64x64 | 481.7 fps | 297.8 fps | 48.0 fps | 44.9 fps |
| 128x128 | 170.5 fps | 90.9 fps | 41.1 fps | 33.9 fps |
| 256x256 | 47.4 fps | 24.0 fps | 25.3 fps | 16.1 fps |

**Table 2:** *Frame rates for 2D slice fire simulations, both with and without a particle system with 2048 particles.*

Figure 3 shows a side by side comparison of the resulting fire for full 3D and 2D slice simulation. Performance-wise, the 2D slice simulation with two slices at 64x64 clearly outperforms the full 3D simulation at 32x32x32 without compromising the visual quality. In fact, the fire may even appear more detailed because of the higher grid resolution used in the individual slices. The fire has a flickering and turbulent behavior and a lot of low level details. This is representative of a raging bonfire, whereas small camp fires and candle flames usually are smoother.

The number of particles used when visualizing a fire simulation affects the performance and this is illustrated by the results presented in table 3. The number of particles has been

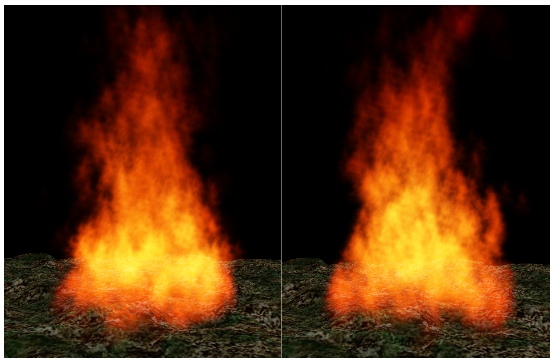

**Figure 3:** *Visual results from 64x64x2 slice (left) and 32x32x32 3D (right) simulations, with 2048 particles.*

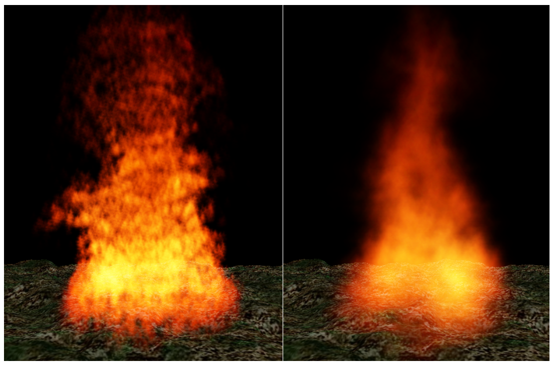

**Figure 4:** *Particle system with 16384 particles (left) and 256 particles (right).*

varied when running 2D slice simulations with two slices of dimension 64x64. The particle size was the same for all tests resulting in very high fill rate requirements for high particle counts. The first column shows the results from performing the fire simulation and the particle system simulation (updating positions, velocities, and colors) but not rendering the particles. In the second column the particles are rendered as well. As can be seen, visualization is the main bottleneck of the algorithm due to the high fill rate requirements when rendering a large number of particles. The simulation on the other hand is pixel processing limited. Figure 4 shows the result of the particle system visualization at two particle counts. The size of the particle splats has also been varied to compensate for the different particle counts.

| Particle count | No rendering | Rendering |
|---|---|---|
| 256 | 471.5 fps | 221.4 fps |
| 1024 | 471.5 fps | 87.8 fps |
| 2048 | 457.8 fps | 48.1 fps |
| 4096 | 443.9 fps | 25.6 fps |
| 16384 | 374.3 fps | 6.7 fps |

**Table 3:** *Frame rates for two 2D slice fire simulations of size 64x64 and a particle system simulation with and without rendering.*

### 4.2. Comparison with other approaches

In this section, we briefly compare our fire visualization against the visual elements and performance of other approaches for visualizing fire. We have defined a set of criteria to guide the visual comparison. These criteria are flickering, turbulent behavior, color, texture, and smoke.

Flickering and turbulent behavior as well as a smooth appearance with realistic colors is best achieved by the offline method presented in [NFJ02]. The focus in [MK02] is on simulation and their fire is not visually convincing. The fire in [KW05] fails in capturing the flickering and turbulent elements as convincingly as ours, in addition to having quite saturated colors. Like our fire visualization, the one in [WLMK02] has some low level texture and realistic colors. As opposed to our work, both [NFJ02], [KW05], [MK02], and [WLMK02] include smoke.

[MK02] achieve 20 fps with a 20x20x20 simulation grid. Using the same grid size, we achieve a frame rate at around 360 fps, mainly because we have implemented the complete fire simulation on the GPU.

When using a grid size of 32x32x32 on a NVIDIA GeForce3 Ti 200, [WLMK02] achieve a frame rate of around 215 fps for the fire simulation alone, and 65 fps when rendering 100 texture splats. At the same grid size we achieve a frame rate of 94 for the fire simulation alone, and are able to render 512 particles at 65 fps. Obviously, the LBM implementation is a lot faster than the stable fluid solver, although the accuracy of the former is uncertain. Since they render the display primitives back-to-front, they use more computation power during the rendering step as sorting is needed.

Finally, [KW05] achieve a frame rate of 190 fps on a NVIDIA GeForce 6800 GT when using two 2D simulations at 64x64 each and extruding them to a full 3D volume. In comparison, we achieve a frame rate of 480 with the same grid size. However, we do not compute pressure templates, nor do we explicitly extrude the 3D volume.

### 4.3. Limitations

Our algorithm for simulating and visualizing fire has several limitations. First of all, the fire is non-interactive in that it does not react to its environment. The fire will thus not be affected by items or obstacles coming in contact with it, which might cause it to look unrealistic.

The characteristic blue core, which appears at the base of small flames and gas flames, is missing from our visualization. As the blue core is not explicitly simulated, we can not visualize it without resorting to ad hoc approximations uncoupled from the physically based simulation. Smoke is basically exhaust gas which has cooled down and is thus supported by the simulation framework. However, smoke visualization has not been implemented yet.

The 2D slice simulation with volumetric extrusion limits the fire to rotationally homogeneous phenomena like torches and bonfires. For other effects, like for example a flame thrower, a full 3D simulation needs to be performed.

There are no high-level options for controlling the fire. Thus, an animator wishing to create a certain behavior and appearance needs to manually experiment with the quite large amount of different parameters for the algorithm. Particle spawn positions must also be synchronized with the fuel gas source field.

## 5. Summary and future work

We have presented an algorithm for simulating and visualizing fire completely on the GPU. The algorithm is based on an underlying fluid simulation coupled with a model of the combustion process. A black-body radiation model and a GPU-driven particle system are used in combination to visualize the result from the simulation.

Future work will include exploring whether volume rendering for visualizing the underlying simulation could produce better visual results or better performance than the current particle system. Another possibility is to explore whether it would be possible to model the interaction between the fire and other objects by incorporating obstacle boundaries in the simulation step. Some work has already been done on fluid and complex object interaction on the GPU [LLW04]. Other possible extensions of the algorithm include blue core simulation and smoke.

## References

[Har04] HARRIS M. J.: Fast fluid dynamics simulation on the gpu. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Fernando R., (Ed.). Addison-Wesley Professional, 2004, pp. 637–665.

[HBSL03] HARRIS M. J., BAXTER W. V., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 92–101.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 115–122.

[KW05] KRÜGER J., WESTERMANN R.: Gpu simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum 24*, 3 (2005).

[Lat04] LATTA L.: Massively parallel particle systems on the gpu. In *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Engel W., (Ed.). 2004, pp. 119–133.

[LF02] LAMORLETTE A., FOSTER N.: Structural modeling of flames for a production environment. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 729–735.

[LLW04] LIU Y., LIU X., WU E.: Real-time 3d fluid simulation on gpu with complex obstacles. In *Pacific Conference on Computer Graphics and Applications* (2004), pp. 247–256.

[Mat97] MATKOVIC K.: *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.

[MK02] MELEK Z., KEYSER J.: *Interactive Simulation of Fire*. Tech. rep., Texas A&M University, 2002.

[NFJ02] NGUYEN D. Q., FEDKIW R., JENSEN H. W.: Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 721–728.

[Ngu04] NGUYEN H.: Fire in the "vulcan" demo. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Fernando R., (Ed.). Addison-Wesley Professional, 2004, pp. 87–105.

[RNGF03] RASMUSSEN N., NGUYEN D. Q., GEIGER W., FEDKIW R.: Smoke simulation for large scale phenomena. *ACM Trans. Graph. 22*, 3 (2003), 703–707.

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

[WLL04] WU E., LIU Y., LIU X.: An improved study of real-time fluid simulation on gpu. *Journal of Visualization and Computer Animation 15*, 3-4 (2004), 139–146.

[WLMK02] WEI X., LI W., MUELLER K., KAUFMAN A.: Simulating fire with texture splats. In *VIS '02: Proceedings of the conference on Visualization '02* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 227–235.