NTNU
Innovation and Creativity

Implementing LOD for physically-based real-time fire rendering

Lars Tangvald

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Torbjørn Hallgren, IDI
Co-supervisor: Odd Erik Gundersen, IDI

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

Problem Description

Objective is to continue earlier work done on physically-based fire rendering by Samuel Grødal and Geir Storli in spring 2006.
Specific problems include:
1. Making the fire part of a large scene by including it in a scenegraph.
2. Implementing level of detail (LOD) for the fire to increase efficiency of rendering
3. Modify fire rendering to take advantage of new features found in Nvidia's Geforce 8 line.
4. Extensive testing of implemented methods to evaluate performance and possible improvements.

Assignment given: 20. January 2007
Supervisor: Torbjørn Hallgren, IDI

Abstract

In this paper, I present a framework for implementing level of detail (LOD) for a 3d physically based fire rendering running on the GPU. While realistic fire rendering that runs in real time exists, it is generally not used in real-time applications such as game, due to the high cost of running such a rendering. Most research into the rendering of fire is only concerned with the fire itself, and not how it can best be included in larger scenes with a multitude of other complex objects.

I present methods for increasing the efficiency of a physically based fire rendering without harming its visual quality, by dynamically adjusting the detail level of the fire according to its importance for the current view. I adapt and use methods created both for LOD and for other areas to alter the detail level of the visualization and simulation of a fire rendering. The desired detail level is calculated by evaluating certain conditions such as visibility and distance from the viewpoint, and then used to adjust the detail level of the visualization and simulation of the fire.

The implementation of the framework could not be completed in time, but a number of tests were run to determine the effect of the different methods used. These results indicate that by making adjustments to the simulation and visualization of the fire, large boosts in performance are gained without significantly harming the visual quality of the fire rendering.

Preface

This is a master's thesis for the Master of Science in Technology (Computer Science) program at the Department of Computer and Information Science (IDI). The thesis was written by Lars Tangvald during my 5th and final year at the Norwegian University of Science and Technology (NTNU).

While working on this thesis I have, together with my supervisor Odd Erik Gundersen, written and submitted a work-in-progress paper to Theory and Practice of Computer Graphics 2007. The paper was accepted for presentation at the conference, and is appended at the end of this thesis.

I would like to extend a thanks to my supervisor Odd Erik Gundersen for motivation, guidance and feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computer hardware becomes more powerful, graphical applications such as games become more and more advanced, becoming increasingly detailed and realistic in their portrayal of virtual worlds. An important element in making a virtual environment seem realistic is the incorporation of natural phenomena such as realistic lighting, water and fire. However, while the detail level of lighting and water in real-time applications has increased radically in recent years, the quality of fire has not. Fire is still mostly rendered using static animations or simplistic particle systems, which is becoming increasingly contrasted with the quality of other natural phenomena. A possible reason for this is that running a more realistic fire rendering is costly compared to the increase in visual quality gained. Realistic rendering of fire requires considerable resources, making it unsuitable for inclusion in a scene with numerous other objects that may also require considerable resources. The focus of this thesis is to create a fire rendering that minimizes the resource usage when the fire is not the dominating object in the view.

1.1 Motivation

While numerous physically based fire renderings have been presented in recent years [DQN02][SR06][YZ03], realistic fire has yet to be seen in real-time graphical applications that contain complete virtual environments. As work on producing fast and realistic fire has been focused only on the fire itself and not on making it part of a larger scene, including such fire in an application would have a severe impact on the application's performance. Thus the most obvious use of virtual environments, games, still only use simplistic fire rendering such as static animations and simplistic particle systems. Examples of fire in games can be seen in Figure 1.1.

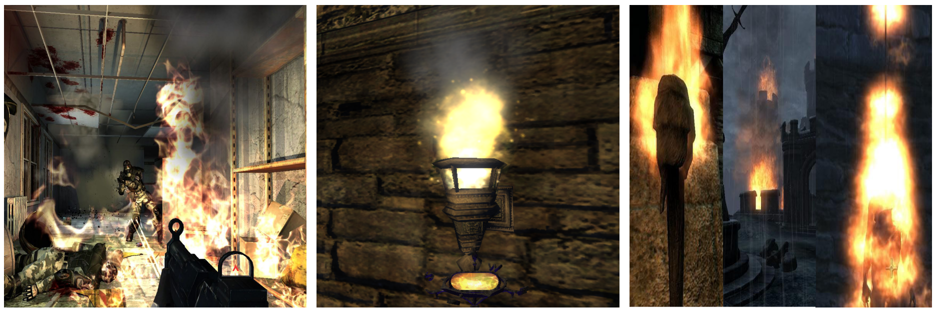

Figure 1.1: Fire rendering in games. Left: F.E.A.R.^{TM}Middle: Thief: Deadly Shadows^{TM}Right: TES: Oblivion^{TM}

This thesis is based on my work from the fall 2006 project "Level of Detail for Physically Based Fire Simulation" and the 2006 Thesis "Physically Based Simulation and Visualization of Fire in Real-Time using the GPU"[KESR06] by Knut Erik Samuel Rødal and Geir Storli. In the fall 2006 project I presented a method for implementing level of detail for a 2D physically based fire rendering also presented by Rødal and Storli. The method was used in simple tests, which indicated that it could give good results, but was not implemented for a real fire rendering. I wish to implement the method for a full fire rendering running on the GPU to verify the validity of the results. I also wish to expand the method from two to three dimensions, and create an implementation for cutting edge graphics hardware to get the best results. The 3D fire rendering presented in [KESR06] and [SR06] will be used as a basis. Additionally, I wish to explore the possibilities offered by Nvidia's newest graphics chip, the G80, to further improve the fire rendering.

1.2 Goals and requirements

The main objective is to present a method for making a physically based fire rendering running on the GPU more efficient by adding dynamic level of detail (LOD) to it. The dynamic LOD should continuously alter the detail level of the fire according to its importance in the current view, thus increasing efficiency without hurting visual quality when the fire is not important to the current view. A framework will be constructed which combines the physically based fire rendering with the LOD method to allow the fire rendering to be rendered in a scene with other complex objects. The requirements used to evaluate the framework are outlined below.

- **R1:** The framework should allow a 3D physically based fire rendering to run in real time.
- **R2:** The framework should allow the fire rendering to be contained in a scene that also contains numerous other complex objects.
- **R3:** The framework should allow several fire renderings to exist in the same scene, each with individual detail levels.
- **R4:** The framework should allow the fire simulation to be run at varying detail levels to increase either performance or quality as needed.
- **R5:** The framework should allow the fire visualization to run at varying detail levels to increase either performance or quality as needed.
- **R6:** The framework should dynamically alter the detail level of the fire to conserve resources while giving the greatest possible quality when the fire dominates the view.
- **R7:** The framework should, as far as is possible, run on the GPU.

The success of the method will be evaluated based on how well the framework satisfies these requirements.

1.3 Approach

My approach for this thesis is outlined below.

- I will examine subjects relevant for implementing LOD for a GPU-based 3D rendering of Fire. The subjects include general background information concerning LOD, architecture of GPU chipsets, GPU programming, methods for rendering fire on the GPU, and a look at which factors affect the performance of an application running on the GPU.

- I will examine previous work done with LOD. I will examine methods that may be used or adapted for use with fire rendering, as well as methods for dynamically adapting the detail level of particle systems, dynamics systems, and GPU programs.

- I will propose different methods for implementing LOD for a 3D rendering of fire running on a GPU. Different methods will be presented for the visualization and simulation of the rendering. The advantages and disadvantages of each method will be evaluated, and a subset of the methods will be chosen for implementation.

- I will implement the chosen methods for a 3D fire rendering running on a GPU. The implementation will focus on allowing multiple fire objects with individual simulations, visualizations, and detail levels, as well as allowing an arbitrary number of other objects.

- I will run a set of tests designed to evaluate the implementation's impact on performance and visual quality.

- I will evaluate the implementation, based on the results from testing. I will also present possibilities for future improvements of the LOD application.

Each item in this list will be presented in a separate chapter of the report.

1.4 Structure

The report consists of the chapters described below.

1. **Introduction:** Describes the motivation and objectives for the thesis, how the problem will be approached, and the overall structure of the report.

2. **Background:** Examines subjects relevant for the thesis. Gives an introduction to GPU-based fire rendering, application performance, GPU architecture and programming, fluid dynamics, and LOD.

3. **Previous work:** Examines previous work done with LOD, with a focus on methods that may be used or adapted for a GPU-based rendering of fire.

4. **LOD for fire rendered on the GPU:** Presents different possibilities for implementing LOD for a GPU-based fire rendering. Evaluates the advantages and disadvantages of each methods, and selects a subset for implementation.

5. **Implementation:** Describes the implementation of the chosen solution.

6. **Results:** Shows results attained from testing of the implementation.

7. **Discussion:** Evaluates the results of the implementation and makes a conclusion based on the evaluation. Discusses possible improvements and future work.

The background and previous work chapters (Chapter 2 and 3) are revised and expanded versions of the background and previous work chapters from the 2006 fall project report. The biggest changes for this thesis are moving the method from 2d to 3d and from the CPU to the GPU.

1.5 Summary

Fire in real-time applications such as computer games has seen few real advancements in recent years compared to other natural phenomena such as water. I attempt to address this issue by implementing a dynamically changing level of detail in order to allow for a realistic fire rendering while minimizing the resource usage when the fire is not important for the view. I look at different methods that can be used or adapted for this purpose, and create an implementation running on a modern GPU. A set of requirements are presented and will be used to evaluate the results of the implementation.

Chapter 2

Background

Numerous fields are related to fire rendering, GPU programming and LOD. To understand fire rendering it is necessary to understand the concept of computational fluid dynamics (CFD), which is the basis for the simulation portion of a physically based fire rendering. There are also different ways to perform both the simulation and visualization of fire.

Performing the fire rendering on a graphics processing unit (GPU) is a possible method for increasing the speed of the rendering while making the CPU free to perform other tasks. The inherent parallelism of GPU architecture can also lead to significant efficiency increases for suitable problems. Writing code for a GPU can be quite different from writing code for a CPU, but there are analogies that can be drawn between regular programming resources and those of a GPU. These analogies can be exploited in order to move problems far outside the field of graphics over to a GPU with good results.

Finally, a brief introduction of level of detail (LOD) is given. To optimize the performance of a graphical program it is important to know what factors play a role in the performance. Such factors can be graphical, in terms of what we see on the monitor, or they could be computations that are performed behind the scenes. Once these factors are known and understood, it becomes possible to find the best way to alter the detail level of a scene in order to get the optimal relationship between performance and quality.

2.1 Computational fluid dynamics

The field of computational fluid dynamics (CFD) is dedicated to using computer calculations to solve fluid dynamics problems [HV95]. Originating in the 1960s, the field has become increasingly popular as the processing power of computer hardware increases. More powerful hardware allows for simulating more detailed models. A CFD program consists of three main parts: pre-processing, solver, and post-processing. The Navier-Stokes equations are most commonly used when implementing a CFD program.

2.1.1 Structure of CFD programs

A CFD program consists of three main parts: pre-processing, solver, and post-processing. The three parts are fairly independent in that changing one part does not affect the others. The most common way of structuring the program is outlined here:

The pre-processing portion of the program uses the program input to define the problem in a way the solver can understand. The general steps of doing this are outlined below.

- Defining the computational domain.
- Forming a grid of discrete cells from the domain.
- Choosing which phenomena to include in the simulation.
- Defining the properties of the fluid.
- Specifying boundary conditions.

The solver takes the output of the pre-processing step as input, and uses it to produce a solution by the steps outlined below.

- Approximating unknown flow variables.
- Discretizing by using the approximations in the fluid equations.
- Solving the equations.

Solving may be handled in a number of different ways, but the most common method today is the finite volume method[HV95]. Finally, the post-processing step consists of analysis and evaluation of the results from the solver. Post-processing often includes some form of visualization, from simple graphs to the detailed visualizations used for realistic fire rendering.

2.1.2 Navier Stokes

The Navier-Stokes equations describe the motion of liquids and gases [HV95]. The Navier-Stokes equations can be used for a variety of different purposes, such as aerodynamics, fluid flow and semiconductors. The governing principle of the equations is conservation of momentum. Rather than model values such as pressure and velocity, the Navier-Stokes equations model the flux of these values. Directly solving these equations will usually be far too time-consuming to be practical or even possible. Therefore, simplifications are usually made by disregarding certain fluid properties such as viscosity, using numerical approximations, and using simplified models corresponding to fluid behavior found by empirical study. The Navier-Stokes equations for incompressible fluids are shown in Equation 2.1.

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{F} \tag{2.1}$$

Equation 2.1 describes the velocity field of the fluid, governing the motion of density through the fluid. The density ρ of the fluid describes how much fluid is contained in a volume unit. The viscosity μ describes how thick the fluid is. The pressure p describes with how much force the fluid is compacted at a certain point. The velocity vector $\mathbf{u}$ describes in what direction and how fast the fluid is flowing at a certain point. $\mathbf{F}$ describes what external forces are applied to the fluid at a point or the fluid system as a whole.

2.2 Fire

Modeling of fire is modeling of gases. When fire is ignited, there is a chemical reaction between oxygen in the air and the fuel source, creating heat [RB98]. The characteristic glow we associate with fire is the result of heat radiation from the gases of the fire system. As the particles of the gas cool down, they become smoke.

A physically based fire rendering is a fire rendering which is based on the physical characteristics of fire, rather than simply being something that looks superficially similar. The type of fire rendering considered here has a separate simulation and visualization part, so the simulation is independent of the visualization and vice versa. Both the simulation and visualization can be done in a number of different ways. A brief description of some of the different methods is given before a more indepth presentation of the approach done in [KESR06].

2.2.1 Different approaches to fire rendering

An important factor in fire rendering is the requirements in terms of performance and accuracy. Whether or not the fire needs to render in real time determines what sort of simplifications and models are used. Running the rendering on a GPU may improve performance, but not to a degree where a fully realistic rendering can be run in real time on current hardware. Different methods give different results in terms of performance and accuracy.

Fire simulation

The simulation of the fire is the computations concerning the motion of the fire, whether through calculating the speed and direction of particles or calculating the values of a velocity or temperature field. One way of simulating the fire is to not really simulate it, but simply use a motion model which appears similar to that of real fire. Using a model is a cheap way to get a fire that appears to behave accurately. Fractal perturbations may be used to emulate the turbulent behavior of fire, but the fire would not be realistic physically, and it is very difficult to include variations such as wind and obstacles [KP89]. A slightly more accurate model would be the use of predefined velocity fields, made to be similar to real fire [PB01]. However, the conditions and parameters of a fire can have a large impact, and seemingly insignificant changes can drastically alter the results, making this approach inaccurate unless all the fires modeled are very similar. A more accurate approach is to use CFD simulations [DQN02]. The fluid dynamics are then calculated continually so the fire develops in a more realistic manner, with the possibility to adjusting the accuracy and performance by using models at low levels.

Fire visualization

The visualization of the fire is what directly affects the view of the scene. In addition to the appearance of the fire itself, effects the fire has on the environment such as lighting are part of the fire visualization. One way of visualizing the fire itself is by using a particle system [AL02][DT03]. If the particles are simple points, this approach requires a huge number of particles to give a good result, preventing real time rendering. Instead, primitives are rendered with a texture and blended together, requiring fewer particles to avoid gaps in the fire [SAKR00]. The downside to using textured primitives is that with larger primitives the clean lines and contours of a real fire are lost, while with smaller primitives more are required, decreasing performance. Another approach to the visualization is to use volume rendering, visualizing the volume of the fire instead of many small particles contained in the volume. Different approaches to volume rendering exist, including mesh rendering [MB06], ray casting [KESR06], and texture splatting [YZ03].

2.2.2 Storli and Rødal's approach to fire rendering

The focus of this thesis is the fire rendering presented in [KESR06]. In their thesis, Samuel Rødal and Geir Storli created a fire rendering that includes many important features such as wind, smoke and dynamic lighting. Their fire rendering does not match real fire visually, but has very good performance and good physical and visual accuracy. Some example screenshots of the fire rendering can be seen in Figure 2.1

Fire simulation

The fire simulation is done by using a CFD implementation with a stable fluid solver presented in [Sta99]. The most important simplifications made are that the fluid is incompressible and has zero viscosity. An incompressible fluid does not expand or contract, while zero viscosity means it has no resistance against motion. While these assumptions make the equations much easier to solve, there is an added problem in that the expansion of hot gases is an important aspect of real fire. To compensate for this problem, expansion is emulated by adding more fluid to the expanding area.

Turbulence is included to create the chaotic behavior characteristic of fire, with vorticity confinement[JK05] used to emphasize turbulence. A global

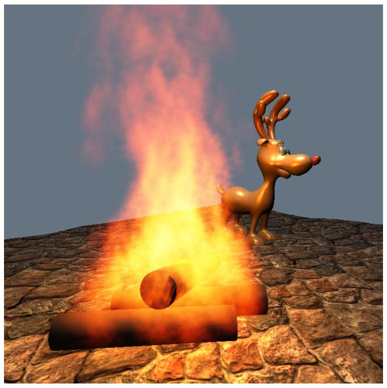

Figure 2.1: Physically based fire rendering by Samuel Rødal and Geir Storli.

wind field is used to give the effects of wind, but also to make the fire react accurately when the fire is moved. The smoke from the fire is modeled together with the fire itself, as the smoke is simply cooled exhaust gases.

The fire is simulated either in 3D or 2D. When simulated in 2D, a slice system is used to expand the result to 3D. By simulating two or more slices, the results can be interpolated out to fill the 3D volume. But while this increases performance while still giving a good visual result in general, it becomes impossible to accurately include the effects of wind on the fire [KESR06].

Fire visualization

Rødal and Storli implemented three different ways to visualize the fire, outlined below.

1. **Black-body radiation:** Uses the density and temperature fields to give a color value for each cell in the grid. Only used in the two-dimensional fire rendering, but looks highly realistic.

2. **Particle system:** Uses a system of textured particles. Each particle is a quadratic polygon rotated to face the camera and textured with a predefined texture. The color and opacity of a particle is decided by the fire heat at the particle's location in the grid. At all times there is a predefined number of particles in the fire. Each particle is spawned around the base of the fire and travels through the velocity field of the fluid, constantly changing its color and opacity. When

a particle's opacity drops beneath a certain threshold, it is removed from the system and respawned at the base of the fire. The smoke is visualized in the same way, albeit with fewer and larger particles with their own predefined texture.

3. **Volume rendering:** Uses a technique known as ray marching. For each pixel, a ray is calculated from the pixel position toward the fire. Rays crossing the fire's bounding box are given color from the grid cells they pass through. However, the performance is worse than for the particle system, and the visual results are not convincing.

Examples of the three visualization methods can be seen in Figure 2.2. Rødal and Storli also included dynamic lighting in their fire rendering. Having the fire light up the environment is an important aspect of making it appear realistic. The lighting effect is created by having stationary or moving dynamic lights inside the fire volume. The intensity of each light is varied according to the heat at the light's position. This lights up the environment, and the varying intensity of each light produces the characteristic flickering effect common to real flame.

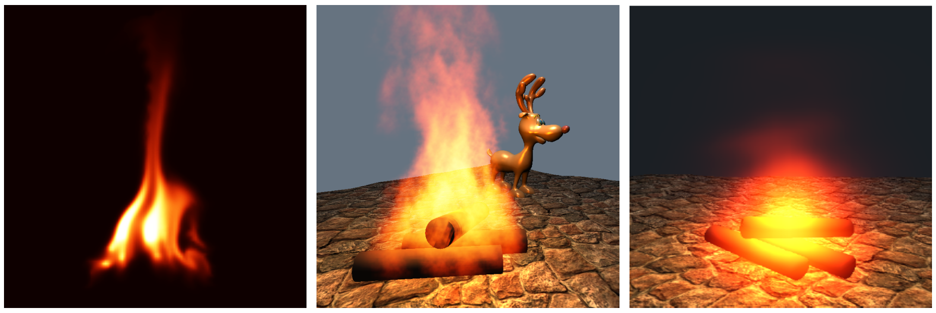

Figure 2.2: Visualization of fire simulations from [KESR06]. Left: Two-dimensional Black-body radiation. Middle: Three-dimensional particle system. Right: Three-dimensional volume rendering.

2.3 Application performance

The processing power required to render a scene depends on the complexity of the objects to be rendered. Simple objects with few details are less computationally demanding than complex objects. However, when an object is far away, or barely visible, it becomes difficult to notice the difference between simple and complex objects of similar shapes. The idea behind LOD is to take advantage of this similarity by reducing the detail of objects that are considered to be less important based on such things as size on screen, distance from camera, motion, and so on [Hue03]. Reducing the detail level of less important objects increases the efficiency of a program relative to the perceived detail the user experiences.

2.3.1 Performance factors

Different components of an application affect the performance in different ways. In order to increase the efficiency of a program it is important to determine which factors affect the performance most, and which of these factors can be safely reduced when the object is not important for the overall appearance of the view. The two different types of factors are considered, analogous with the fire rendering described earlier. Simulation factors are not directly related to the appearance of an object, but include such things as object dynamics and artificial intelligence (AI). Visualization factors are directly related to the appearance of the object and include such things as texturing and lighting.

Simulation

The simulation portion of an application consists of all the processes not directly related to the visual result. Simulation may cover different aspects of a scene, from physics calculations to procedural animation [TG00]. Two of the most important simulation factors can be said to be AI and physics:

- **AI:** Determines the behavior of actors based on certain rules and the current state of the application. The complexity of AI calculations may vary greatly, from a moving paddle in Pong[1] to the behavior of human characters in a detailed simulation of crowd behavior[CO02]. AI has become increasingly complex in recent years, with game developers in particular attempting to create the most lifelike adversaries possible, from first-person shooting games to turn-based strategy games.

[1]http://www.pong-story.com

- **Physics:** has always been very important for certain application types. In the oil industry, the accuracy of a simulation result may decide whether a venture results in a huge profit or staggering loss. In construction, physical simulations are used to determine the strength of materials needed for a certain structure. For a real-time application the desire for accurate physics must be weighed against the need for speed. However, as processing power increases, the accuracy of physics also increases, resulting in games with physics that appear to be highly realistic, and even fairly realistic fire rendering that runs in real time.

The impact of these factors run the gamut from insignificant to dominating, but play an important role in modern applications. The simulation portion of a common application typically runs on a general CPU rather than specialized hardware. As general hardware is slower for most any task than hardware specifically designed for the task would be, simulations lose some performance compared to tasks that run on specialized hardware. This is changing, however. With ways being found for simulation code to make use of the advantages of GPU chipsets, such as greatly increased parallelism. Additionally, specialized hardware is starting to appear for physics processing[2].

Visualization

The visualization portion of an application consists of displaying the current state of the application in some manner for the user. This visualization does not strictly have to be very visual (sound is another example), but the visual portion is the focus here. A typical visualization may include several factors as outlined below.

- **Geometry:** There are a number of different ways to represent geometry visually. In older games, geometry was usually represented with two-dimensional sprites; simple images, possibly with several images to represent the object seen from different angles. 2d sprites is a fast but visually unconvincing way to visualize an object (though still sometimes used for certain objects such as fire in a game). Another way is to use volume rendering. An object is seen as a collection of cells, each having color and opacity values. The object is rendered either by some form of ray tracing, combining the values of each cell corresponding to a

[2]http://www.ageia.com/

pixel on the monitor, or by extracting a surface from the object [Blo88]. Volume rendering is seldom used in games, but is more common in scientific applications, as it gives a good representation of the content of the object and not just its shape. The most common rendering method for real time applications is polygonal rendering. The surface of an object is approximated by a series of small patches, commonly triangular or quadratic. The accuracy of the representation depends on the number of polygons used. Modern GPU chipsets are optimized for polygonal rendering, making polygons a fast method which dominates real time graphical applications such as games.

- **Textures:** Texturing is a way to increase the apparent detail level of an object without increasing geometry complexity. A texture may include information about color, materials, surface direction and more. Using textures, a flat polygon may appear to have a large amount of detail. While textures increase the calculation complexity of a scene, it is vastly faster than trying to visualize the same detail level using only geometry. A brick wall rendered using only geometry would require a very large number of polygons for every brick in the wall. Using textures, the same wall could be rendered using only a single textured polygon.

- **Lighting:** Lighting is a very important factor in making a scene appear realistic. The lighting model used has a large impact on the appearance and complexity of a scene. A simple lighting model might simply calculate the lighting for a point based on the direction of the surface compared to the direction of the light, whereas more complex models might include the effects of light bouncing off objects or the tracing of light rays as they pass through and bounce off different surfaces. The latter two, known respectively as radiosity[LG96] and ray tracing [Shi90], are very costly to calculate, and typically not used in real time applications.

- **Resolution:** The resolution of different aspects of the scene may have a large impact on performance. The resolution of the entire scene decides how many pixels need to have their color calculated. Low resolutions are faster to render, but the result looks more blocky. The colors of each pixel is a sample from the application scene. When the number of samples is low, the discrete nature of each pixel becomes more apparent. This effect of distortion due to a low sample count is known as aliasing. The effect of aliasing may be reduced with a number of different techniques, but this increases calculation complexity. In addition to the resolution of the view frame, the resolution of texture images or grids used for volumetric calculations have much the same effect, with greater resolutions giving more accurate results at the cost of increased calculation complexity.

These factors can complement each other in terms of complexity and realism in different ways; more detailed texturing may reduce the need for detailed geometry, but at lower resolutions it is harder to see fine detail in objects, so their detail level does not have to be as high.

2.3.2 LOD

For any given view of a scene with many different objects in it, it is likely that some objects are more important for the appearance of the scene than others. Objects that are closer to the camera appear larger than those further away, and it is harder to discern fine detail on the objects far from the camera. The basic premise being LOD is to take advantage of the fact that when an object is far away or obscured in some way, a less complex object will have the same appearance to the viewer while requiring less processing power to render [Hue03]. By replacing the objects that are less important to the view with less detailed versions of themselves, the performance of the application can be improved with little or no impact on the visual quality of the scene.

LOD in graphics

The goal of using LOD for graphical objects is to adjust the detail level of less important objects to reduce the complexity of a scene while retaining their overall appearance. LOD for graphical objects is generally divided in three separate methods, as outlined below [Hue03].

- **Discrete:** When or before the program is initialized, several versions of an object are created. Each version of the object has a different detail level, such as different number of polygons for a geometric object. When the scene is rendered, the application chooses one version of the object depending on how important it is for the current view. If the object is of little importance, a version with a low detail level is chosen, and so on. Discrete LOD is easy to implement and fast, but since each version is stored separately, more storage space is required. Also, when an object transitions from one detail level to another it can be noticeable, leading to a jarring popping effect. Additionally, if an object is very large rendering the entire object at a single detail level may give poor results, since one part of the object may be important for the view, while another is relatively insignificant. An example of a polygonal model displayed at two different detail levels can be seen in Figure 2.3[3].

- **Continuous:** Only stores a single version of the object. Changes in detail level are applied dynamically during run-time. A suitable detail level is chosen based on the object's importance. The structure of the object is then changed according to some predefined methods. This will usually give much smoother transitions than with discrete LOD, as the transition steps can be as small as desired. Continuous LOD is slower, however, and like discrete LOD uses a single detail level for the entire object, no matter how large an area it spans.

- **View-dependent:** Approaches the problem in a different way, by only considering the view from the camera. The scene is then altered by reducing detail levels as much as possible while preserving the appearance from the camera. A sphere, for instance, might be changed so that the front and back are simplified greatly, while the profile from the camera's point of view is largely unaltered. Unlike discrete and continuous LOD, view-dependent LOD is not restricted to a single detail level for a single object, making it more suitable for large objects such as terrain. View-dependent LOD requires significantly more processing power than discrete or continuous, but can give much better results, especially for objects spanning large areas.

LOD for dynamic systems

LOD is not restricted to visualization. LOD can also be applied to dynamic systems [JH97]. Calculations can be performed with varying degrees of accuracy, with less accurate calculations used when an object is less important for the current view. There is an additional consideration that needs to be made,

[3]Model is taken from the modeling program Blender (http://www.blender3d.org).

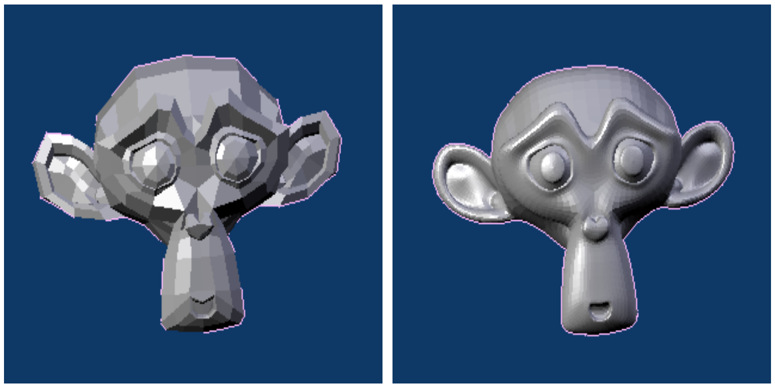

Figure 2.3: Polygonal model of a monkey displayed at two different detail levels.

however. In visualization, the detail level of the current frame only affects the current frame. For a dynamic system, however, simplifications made to the current frame may affect future frames as well. When a simplification is used, it introduces an inaccuracy in the result. This result will generally be used for the calculations in the following frames, and if the following frames also uses simplified calculations the inaccuracy will be increased with each step, potentially producing a very different final result from what is correct.

This may not be as large a problem as it seems, though. The main concern for most real-time applications is how the user perceives the result, and not how accurate the result actually is [Che01]. Therefore, flaws in the calculations are generally acceptable, as long as the user does not notice that they are flawed.

Another problem which does not exist for visualization is what to do when a moving object is not visible at all. Consider a ball that is thrown in the air. If the viewer turns away from the ball, the application should use as few resources as possible to calculate the motion of the air, since that motion won't be visible. However, the ball must continue moving so that when the viewer turns back to look at it, it appears to have followed its trajectory correctly. If the ball is still hanging in the same position as when the viewer turned away it will ruin the illusion of accuracy. Approximating the position of the ball when the viewer turns back is therefore important to maintain the illusion while reducing calculation complexity [DF97].

2.3.3 Application areas of LOD

Because the primary reason for using LOD is to improve performance, it is mostly used for applications designed to run in real time. Interactivity is also often a factor, since non-interactive applications are far easier to optimize. Using LOD allows for including more detail without sacrificing overall performance or improving performance without sacrificing apparent quality. Modern computer games often include so much detail in a scene that most home computers would not be capable of rendering it all in full detail. But since user experience is more important than actual realism and accuracy the most important aspect becomes to make the scene appear detailed and realistic. A few possible uses for LOD algorithms are outlined below.

- **Geometry:** Alteration of object geometry is probably the most widespread use of LOD. When an object is far away or otherwise unimportant, it becomes harder to see much more than the object's general shape. The object may then be rendered with less geometric detail than usual, increasing performance without noticeably sacrificing visual quality [SPC05][FD05][Eri00].

- **Terrain:** A subset of geometry, terrain deserves special consideration as it is somewhat different from other geometric objects. Terrain is usually rendered as a single large object that spans a very large area in the scene. This makes traditional discrete and continuous LOD unsuitable for terrain, as the entire model is rendered with a single detail level. Terrain is therefore one of the primary usage areas of view-dependent LOD [SRWH98], and much research has been done on how to render terrain efficiently[Hop98][BDL03][Lev02][PL96][SRWH98].

- **Physics:** Physical calculations can be performed with a number of different accuracy levels depending on the importance of the object. If an object is not clearly visible, it becomes less important for the object to behave accurately and approximations may be used [JH97][DF97].

- **Logic:** Logic here refers to the autonomous behavior of actors that are not controlled by the user. The behavior of non-player characters (NPCs) in a game is an example of the result of logic calculations. These calculations can, like physical simulations, be simplified when the NPC is far away or not visible [CO02][JH97][Che01]. If an object is to move between point A and B and the viewer is watching it all the way, the application calculates each step of the journey and updates the object's position continually. If the viewer is not watching, however, the application only needs to calculate how long the movement takes, and simply change the object's position from A to B.

2.4 Graphics processing unit

In earlier years, graphical calculations would be done on the CPU. A matrix representing the pixels of the screen would be constructed in software, and then sent to the monitor for displaying. The graphics hardware was merely a tool for sending the pixel matrix from the CPU to the monitor. As the hardware evolved, more functions were moved from software to hardware. An implementation in hardware has the advantage of being much faster than the equivalent software implementation and lessening the CPU's burden. The downside is that hardware implementations lack the flexibility of the CPU. This made it hard for software developers to make the most of the hardware without binding themselves to a single manufacturer or even card. The creation of programmable shaders meant that developers had much more power to create the effects they wanted in graphical applications. Most current cards work with two different types of shaders; vertex shaders and fragment shaders, with the newest cards having a third: the geometry shader. Traditionally, the shaders use separate portions of hardware for their functionality. However, new chipsets use a unified shader architecture, combining the functionality into a single unit to make better use of available resources. Programmable shaders also had a secondary (and possibly unexpected) result, in that GPU chipsets could now be used to perform tasks that are not directly related to graphics. Because modern GPUs are focused on parallelism, tasks that are suited for running in parallel will often be much faster on a GPU than on a CPU. Using the GPU for tasks that are not directly related to graphics has become known as general purpose GPU (GPGPU) programming[Pha05].

2.4.1 Architecture

The hardware of a GPU chipset receives data from the parent application, and outputs a frame buffer. Data passes through the different portions of the architecture, with each step using the output from the previous step as input. In most current graphics cards, the vertex and fragment shaders are separate units with separate resources. In newer cards using a unified shader architecture, all shader types share resources, which increases efficiency. In addition to the different shaders, the GPU chipset also contains certain fixed-feature parts such as depth-testing and rasterization. The architecture of an older Nvidia Geforce 6 card can be seen in Figure 2.4[4], while the architecture of a newer Nvidia Geforce 8 card can be seen in Figure 2.5[5].

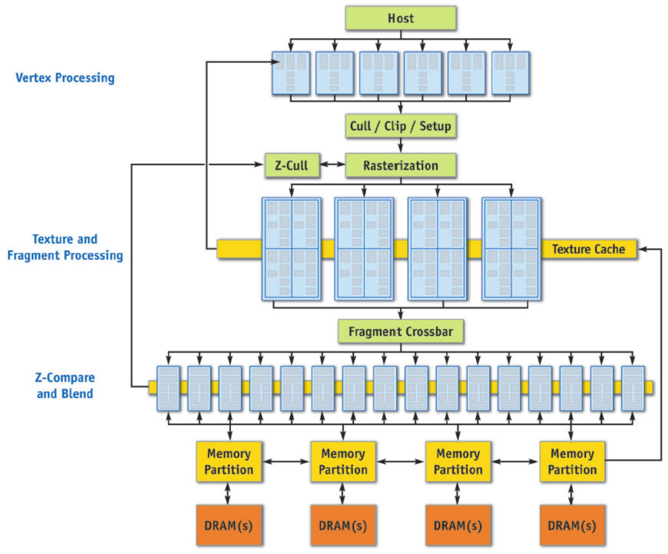

Figure 2.4: Geforce 6 architecture

2.4.2 Vertex shader

The vertex shader takes as its input vertex data from the application, consisting of such things as vertex position, vertex normals and texture coordinates. Its most basic task is to transform these values from the world coordinates of the application to screen coordinates. The vertex shader may also make arbitrary changes to any of these values to change the appearance of the view.

[4]Image source: [Pha05]
[5]Image source: http://www.hardwaresecrets.com

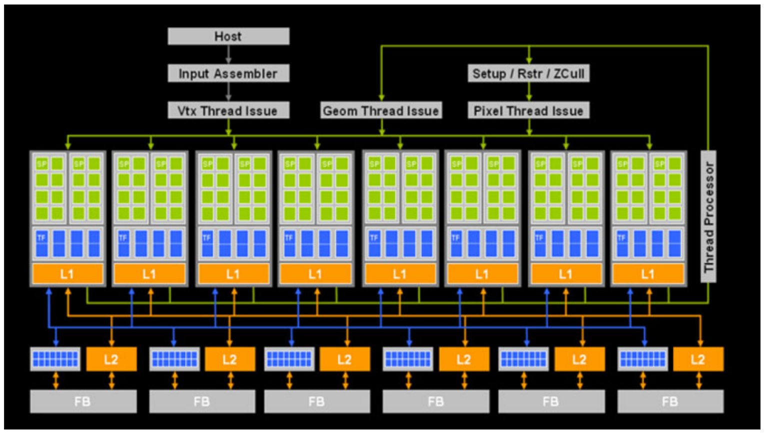

Figure 2.5: Geforce 8 architecture

2.4.3 Geometry shader

The geometry shader is only present on newer GPU chipsets. It takes as its input the output from the vertex shader. Unlike the vertex shader, which can only change existing data, the geometry shader can add or remove primitives from the scene. An example could be the creation of a sphere. With only a vertex shader, the application would need to supply every vertex of the sphere to the GPU. With the geometry shader, the application could simply send the position and radius of the sphere, and the geometry shader would create all the other primitives needed to form the sphere. The advantage is that far less data needs to be sent from the system to the GPU, thus increasing efficiency.

2.4.4 Fragment shader

The fragment shader takes as its input the results of the rasterization of the primitives generated by the vertex and geometry shaders. The output of each fragment shader can be said to be a potential pixel in the finished result. The result from a fragment shader may be discarded if the fragment in question fails its depth test (it was behind another fragment and the two are not blended together in any way). The fragment shader uses the lighting and primitive information calculated in previous shaders to determine the color of a single fragment.

2.4.5 Unified shader architecture

One problem that can appear with the traditional, separate shader architecture is that the amount of work for each shader type may be very different. In many graphical applications the vertex shader does little more than pass data on, while the fragment shader makes advanced calculations for lighting and texturing. The result may be that the vertex shader resources are largely idle while the fragment shader is heavily taxed. The unified shader architecture is an attempt to remedy the problem by combining the resources for the different shader types. The resources are allocated to the shaders according to which shader perform the most complex calculations.

2.4.6 GPU programming

Graphics hardware has in the later years approached and in some ways surpassed the CPU in terms of flexibility and power [Pha05]. A very significant step has been the introduction of programmable GPUs, which allow programmers to do just about any task with the graphics hardware instead of the CPU. This can be a tremendous advantage, as even with hardware that by today's standards is becoming obsolete, the GPU performs certain tasks much faster than the CPU. GPU programs are usually written in one of the three languages GLSL[6], CG[7], or HLSL[8]. The three languages are very similar in syntax and semantics, and porting programs from one to another is usually fairly simple. The syntax and semantics of the three languages are close to that of C, but with their own set of standard routines and data types. The resources available for GPU programs also differ from those of a normal CPU program, but there are certain analogies that can be made between CPU programming and GPU programming, and these analogies may be used to port almost any CPU program to GPU code.

GPU analogies

The resources available when creating GPU programs differ from the resources available when creating standard CPU programs. However, analogies may be made between the resources available to a GPU and the resources

[6]http://www.opengl.org/documentation/glsl/
[7]http://developer.nvidia.com/page/cg_main.html
[8]http://www.neatware.com/lbstudio/web/hlsl.html

available to a CPU to help understanding how to transfer algorithms between the two. The GPU analogies of common CPU resources are outlined below [Pha05].

- **Data arrays:** The primary form of stored data on the GPU is in textures. A two-dimensional texture is basically a two-dimensional array of numerical values.
- **Inner loops:** The inner loop of a CPU program corresponds to a fragment program on the GPU. The fragment program computes the value of the smallest output primitive; the pixel.
- **Computation invocation:** The invocation of the inner loop is the invocation of the fragment program: rasterization.
- **Feedback:** Feedback is not directly available from GPU programs. However, the result does not have to be rendered to a monitor. The results of a GPU program can be rendered back to a new texture, and the values can be extracted from this texture.
- **Computational domain:** The size of the computational domain is determined by the texture coordinates used for rendering. The coordinates can be seen as array indices, and the rasterizer interpolates between the given coordinates to map out the entire range.
- **Computational range:** As the rasterization invokes the computation, the coordinates of the primitive's vertices determine the computation range. As the primitive is rasterized, it's size determines the number of separate fragments calculated.

These analogies are particularly useful when performing GPGPU programming.

GPGPU programming

GPGPU programming is based on using the properties of the GPU to perform computations not related to graphics [Pha05]. While most modern GPUs have been designed with real-time graphical applications, such as games, in mind, the programmability introduced with the shader architecture has opened up many possibilities for using it for other purposes. As an example we examine a mathematical function. The function takes an input of some form, and returns an output. To implement this on a GPU, we need three things: A way to send the input to the GPU, a way to perform the function, and a way to return the result to the user. The input can be passed to the GPU either as a value stored in a variable or, if needed, a series of values stored in a texture. Next, the programmable shaders can be used to compute the result of the function. Finally, the result can be returned by using the GPU's render-to-texture ability.

The above approach can be used for virtually any problem, though because of the relatively high overhead in sending data to and from the GPU, it's best if all the necessary data for the problem can be stored in the GPU's texture memory, which on modern graphics cards is usually 256mb or more.

Chapter 3

Previous Work

The premise behind level of detail (LOD) is to alter the detail level of an object according to its importance to maximize the ration between application performance and accuracy [Hue03]. While little research has been done on how to implement LOD algorithms for physically based fire rendering, a considerable amount of work has been done on creating LOD methods for other areas of application. Some of this work may be used or adapted for use with a fire rendering.

Prior to changing the detail level of an object, a set of rules need to be defined for how the wanted detail level of an object is to be calculated. Numerous different rules may be used, depending on the circumstances of the application.

LOD algorithms may be categorized according to the problems they are used to solve, with different methods being used for different portions of an application. LOD for dynamic systems deal with simplifying the dynamics part of a scene by approximating or culling calculations that do not need full simulation. Particle LOD deals with increasing the efficiency of particle systems by manipulating particle counts or particle interactions. Finally, numerous other LOD methods may be adapted for use with the fire rendering.

3.1 LOD conditions

Prior to applying LOD algorithms to an object, the object's preferred detail level must be determined. The preferred detail level is calculated based on how important the object is for the current view. If an object dominates the view it should be rendered at a high detail level, but if the object is relatively insignificant it is best not to waste resources on it. Determining the importance of an object can be done in a number of ways, outlined below [TKH04].

- **Distance:** The distance from the object to the point of view is generally a good indication of how important the object will be for the current view. If the object is far away from the point of view it is rendered at a lower detail level as fine detail is harder to see at a distance. One problem with this approach is that the parameters of the view changes, distance may be a poor indication of importance. Field of view and focus changes may make a far-away object more important than a closer one.

- **Size:** The size of an object on the current view may be a better indication of importance than distance where view parameters are subject to change. If an object is large in the view, it is more likely to draw attention, and fine detail is easier to discern. Thus, larger objects are rendered with more detail than smaller object.

- **Visibility:** How visible an object is may determine the detail level it is rendered with. If an object is obscured by another, it is generally not necessary to render it at full detail, as much of the object can't be seen. If the object is not visible at all because it is outside the view or completely obscured behind another object, it may not be necessary to render the object at all. Environmental conditions may also affect the visibility of an object. Fog or smoke may reduce the visibility of the object, allowing it to be rendered at a lower detail level without a noticeable decrease in quality.

- **Eccentricity:** When looking at an image, viewers tends to focus first on an area around the center of the image. Objects near the center of the view become more noticeable than those closer to the edges. Objects near the center of the view may therefore be rendered at higher detail levels than objects near the edges.

- **Speed:** If an object is moving, it may be harder to discern fine detail on it. Objects moving at high speeds may therefore be rendered at lower detail levels without the decrease in quality being obvious to the viewer. However, moving objects also tend to draw attention more than stationary objects, and this needs to be taken into consideration, especially if the speed of the object is low.

- **Attention-directed:** Objects that stand out from their environments will tend to be more noticeable than objects that blend in more. If all objects but one in a scene are gray, while the last object is a vibrant color, the last object will stand out, and therefore be more noticeable than others and should be rendered with more detail. Conversely, objects that are very similar to their environments may be rendered with less detail.

Additionally, there may be other factors not directly related to the individual object that need to be considered. These factors may deal with scene environments or application considerations that may affect the detail level of renderings [TKH04].

- **Frame rate requirements:** It may be desirable for an application to run with a constant frame rate regardless of the content of the current view. If the current frame rate is below the target frame rate, the detail level of all objects may be lowered correspondingly, or the impact of the importance rating may be increased, lowering the detail level of unimportant objects more than that of important objects.

- **Limitations of human vision:** Human vision is geared toward noticing certain features in a view. Focus effects tend to make us ignore objects that are out of focus, so if certain objects are rendered out of focus they can be rendered at a lower detail level, both because the viewer will tend to ignore it and because the blurriness will make it harder to see detail. Also, if a certain task is to be performed, we tend to focus most of the objects related to that task to the exclusion of other objects.

Depending on the circumstances of an application, some or all of the conditions presented may be used to determine the detail level of the objects in a scene.

3.2 Dynamics LOD

The CFD portion of the physically based fire rendering is an important factor of its performance. Simplifying the calculations of the CFD might therefore be a good way to increase performance. When an object is not important for the current view, it may be possible to simplify its simulation to increase performance without hurting the apparent quality of the view. Additionally, any object that are out of view may have their simulations culled, drastically reducing the amount of calculations needed.

3.2.1 Simulation simplification

When an object is determined to be of little importance to the current scene, the simulation of that object can be simplified in some way. A way to do this is to replace the simulation with a simpler simulation, which is less accurate but faster to calculate. [JH97] presents an environment with a number of bouncing robots and a single large puck. As the puck and the robots move around, the robots attempt to avoid collisions with the puck, the walls, and each other. Robots that are not important for the current view use simpler simulations to calculate their movements. If a robot seems to be about to collide with something, however, it is switched back to full simulation to achieve an accurate collision response. This increases the overall efficiency of the application without hurting the appearance noticeably.

3.2.2 View culling

View culling is a good way to reduce the computational complexity of dynamic systems. Reducing the calculation detail for parts of the system that are not currently visible is a good way to increase efficiency. In [Che01] view culling LOD is used to increase the efficiency of a traffic system. Full simulation of actors in the system is only done for the actors within the current view. For the rest of the system, approximations are used to calculate the position of actors by using an event system. The vehicles outside the view are moved between predicted locations in discrete steps. If the path of a vehicle intersects the current view, the vehicle is moved in a discrete step to the entry point of the view, and then simulated fully until it leaves the view again.

As for the vehicles described, a view culled simulation will generally be replaced by some form of proxy simulation until the system determines it should be simulated fully again. Proxy simulations may vary from doing nothing at all to attempting to approximate accurate states of the simulation [SC01]. [DF97], [SC98] and [Che01] consider three basic problems that are likely to appear when performing view culling on dynamic systems:

- **Consistency:** If the view turns away from a simulation, and then turns back at a later time, the simulation should appear to have been running fully in the meantime.

- **Completeness:** If fully simulated, objects may move in and out of the view on their own. It is important that proxy simulations allow for predicting when objects should return to being fully simulated. Otherwise, a fixed view may eventually empty itself, as objects move out of the view, but never in.

- **Causality:** There may be relationships between objects that affect the results of the simulation. A simple example may be two objects that move perpendicular to one another, one outside the view and one initially inside. If both objects are simulated fully, the object initially inside the view will leave it, but bounce off the other object moving along its path, and return to the view. However, if the second object has its simulation culled, the collision might not register, resulting in the first object never reentering the view.

The importance of these problems depend on how easy it is for a viewer to spot flaws in the proxy simulation used [DF97]. If it is easy for the user to predict future states of a simulation the proxy simulation should attempt to approximate these future states correctly. If the future states of the simulation are impossible to predict, however, it is much harder to spot inconsistencies, and less effort needs to be made to approximate the simulation. An object moving in a straight line with constant speed is an example of a simulation that is easy to predict, whereas an object with a completely random motion pattern is an example of a simulation that is impossible to predict.

3.3 GPU LOD

Running the fire rendering on the GPU gives a large increase in performance compared to running it on the CPU. However, running an LOD algorithm for the fire on the CPU would remove much of this advantage, so finding ways of running LOD on the GPU is necessary to get a good result.

[JJ05] presents a method for running a geometry LOD method on the GPU. The method implements a form of discrete LOD by storing the structure of an object in a quad-tree, which is then stored in texture memory. The method performs two passes when running: one for LOD testing and one for rendering. In the testing pass, each node of the quad-tree is tested in a fragment program. If the error of a node is below a certain threshold it is kept, otherwise it is discarded. Finally, visibility culling is performed. A second pass renders the geometry of the selected nodes. The method gives better performance than an equivalent CPU implementation and reduces CPU load.

[FR06] presents a method for implementing continuous LOD on the GPU. An object is stored at the highest detail level in GPU memory, and updated according to current LOD parameters as the application runs. When the detail level is altered, the data is altered by the CPU, and then sent to GPU storage, relying on the speed of the PCI-E bus for performance. As such, the LOD algorithm itself does not really run on the GPU, as mesh alteration is done on the CPU and the results uploaded to the GPU.

3.4 Particle systems LOD

Calculating particle behavior can have a significant impact on performance. While calculations for each particle will usually be fairly simple, the commonly high number of particles results in a high total complexity. A common way of reducing the detail level of a particle system is to reduce the number of particles (often while increasing the size of each individual particle). This reduces the number of calculations needed for the system. A different approach is to cluster particles together. In [DO01] a method for clustering particles is presented. The idea behind particle clustering is to increase the speed of simulation calculation without sacrificing visual quality. Particles can be clustered together based on such factors as position and speed. Each particle is visualized individually, but simulation calculations are performed per cluster rather than per particle. All the particles in a cluster share one physical behavior, which can lead to errors when collisions occur, as all particles in the cluster will have the same collision response. The collision problem can be corrected by dissolving clusters that are about to collide, allowing each particle to have a more accurate collision response, and then reform new clusters after the collision. Another problem with the clustering approach is that the cluster structure may be noticeable along the edges of a particle system, as seen in Figure 3.1.

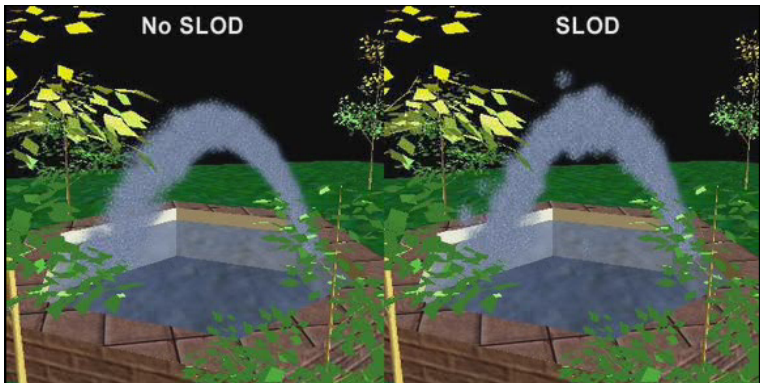

Figure 3.1: Fountain particle system with SLOD implemented. Left: System running without SLOD. Right: System running with SLOD.

3.5 Other LOD methods

There are numerous LOD methods that may be used or adapted for use with the physically based fire rendering. For resizing the CFD grids it may be possible to adapt methods for image filtering, or methods for adjusting the polygon counts of geometric models. Using billboarding to speed up visualization may be an option when the fire is at a low detail level.

3.5.1 Image filtering

In two dimensions, the discrete cells of the grid used for CFD calculations may be considered to be similar to the pixels of a digital image. As such, it may be possible to use methods for resizing digital images to resize the CFD grid, which should have a considerable effect on performance.

Different methods for resizing images are presented in [RCG02]. There are many different methods for resizing images, but the more complex methods are not considered here. The methods considered are outlined below, first for increasing image size and then for reducing it.

- **Pixel replication:** A very simple method for increasing image size if the image is to be resized with an integer factor. Each pixel is copied in both direction according to the resizing factor. Only works when the resizing factor is an integer, and the result is very blocky.

- **Nearest neighbor filtering:** The new image grid is considered as lying on top of the old grid. Each pixel of the new image is then given the same value as the pixel closest to it in the old image. The result has the same problem with blocky appearance as with pixel replication, but the resizing factor does not have to be an integer.

- **Bilinear filtering:** Works similarly to nearest neighbor, but instead of using the nearest pixel value from the old grid, an interpolation between the four nearest neighbors are used. This reduces the blocky appearance from nearest neighbor, but if the resizing factor is large the result will look very blurry.

- **Bicubic filtering:** The most common method for image resizing. Similar to bilinear and nearest neighbor, but uses the nearest 16 neighbors to give value to each new pixel. The results of bicubic filtering are better than for bilinear, with less blur and clearer edges.

Results from using some of these methods can be seen in Figure 3.2[1].

Using these methods for shrinking images works exactly the same way, with simple pixel replication replaces by pixel deletion, where columns are rows of pixels are deleted in accordance with the resizing factor. The danger when shrinking images is in losing fine detail such as edges. With the simpler methods such as pixel deletion or nearest neighbor it is more likely that important detail is lost. Blurring the image slightly before shrinking it is a way to counteract this by spreading the wanted detail out across a larger portion of the image.

[1]Filtering performed in Adobe Photoshop™(http://www.adobe.com)

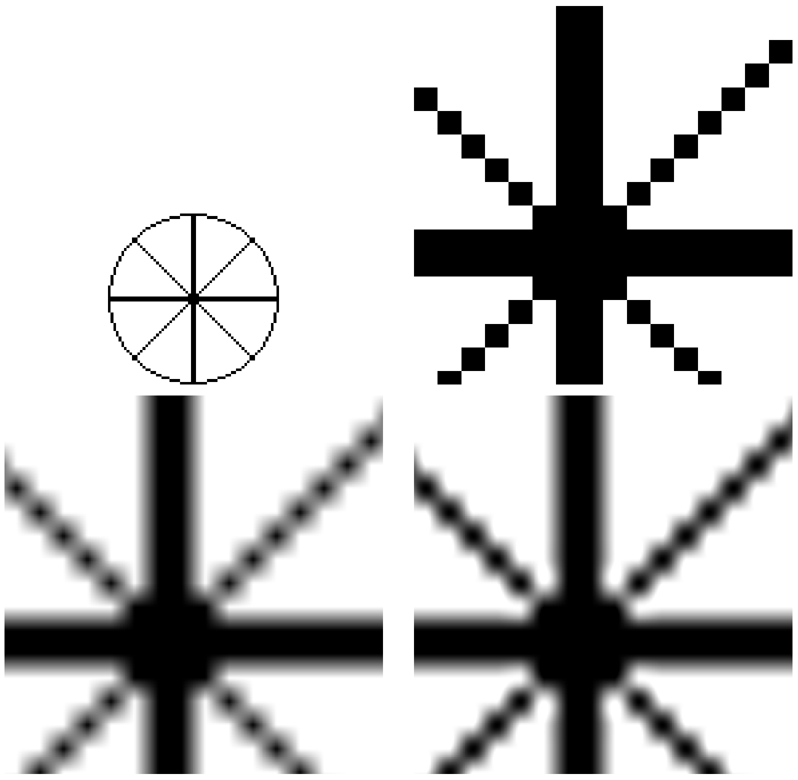

Figure 3.2: Image filtering used for zooming. Image of size 64x64 is zoomed to 1024x1024 using three different filtering techniques. A portion of each zoomed image is displayed. Top left: Original 64x64 image. Top right: Nearest neighbor. Bottom left: Bilinear. Bottom right: Bicubic.

3.5.2 Simplification of polygonal geometry

Polygonal meshes are non-uniform grids with discrete cells. As such, it may be possible to apply methods for dynamically restructuring such meshes to the CFD grid used for the fire rendering. Since polygonal meshes is the dominating method for geometry visualization in real-time applications, a lot of research has been done to determine efficient LOD methods for polygon meshes [SPC05][FD05][Eri00]. The general approach is to combine or remove certain parts of the geometry to increase rendering speed. Some methods that may be adapted for use with the CFD grid are outlined below [TKH04].

- **Vertex removal:** A vertex is removed from the mesh. This effectively deletes all faces connected to that vertex. New faces are constructed over the hole, using fewer faces.

- **Vertex clustering:** A uniform grid is placed over the mesh. For each cell in the grid, every vertex inside the cell is merged, reducing the number of vertices and faces.

- **Vertex pair contraction:** Combines pairs of vertices into single vertices according to some algorithm, usually an error measurement.

- **Edge collapse:** Merges the two vertices of an edge together, deleting the edge and collapsing the edge's faces into edges.

- **Face collapse:** Merges the three vertices of a face together, deleting the face and turning the neighboring faces into edges.

- **Face clustering:** Similar to vertex clustering, but with faces instead of vertices.

These methods may be used for any grid of discrete cells, but because almost all polygonal meshes are irregular the result may not be as good for uniform grids such as the CFD used for the fire rendering.

3.5.3 Billboarding

[SB05] presents a method for increasing performance when rendering large landscapes, using a technique known as billboarding. Billboarding is a method for simplifying detailed geometry by projecting the geometry as textures to a much less detailed geometry. In [SB05], highly detailed models of tree foliage is projected onto a small number of polygons, slanted to give the best view from the camera's position. Approximations of lighting are calculated to increase the apparent accuracy and to help smooth the transitions from one detail level to another. This method allows for rendering highly detailed landscape with a large number of trees and other objects at a good speed.

Chapter 4

LOD for fire rendered on the GPU

When implementing a method for dynamic LOD for a physically based rendering of fire, there are three separate aspects to consider. The three aspects are how to calculate the desired detail level, how to alter the visualization of the fire to reach the desired detail level, and how to alter the simulation of the fire to reach the desired detail level.

This chapter will present a simple overview of the different concepts and approaches used to determine the most suitable methods for implementation. Additionally, different constraints and possibilities introduced by the fire rendering used as a basis will be considered. Different possibilities for the three aspects of the LOD algorithm are then presented an evaluated. Finally, a subset of the presented methods are chosen for inclusion in the LOD framework.

4.1 Overview

There are three separate aspects of the implementation of dynamic LOD for the fire rendering. These three aspects are labeled overall LOD, visualization LOD, and simulation LOD. Overall LOD is basically a value that describes the importance of the fire to the current view, and may be calculated based on different conditions such as distance from camera, visibility, etc. Visualization LOD is using the overall LOD value to change the visualization component of the fire rendering. If the overall LOD value is lowered, the visualization is rendered using a lower detail level, increasing the speed of the rendering. Similarly, simulation LOD uses the overall LOD value to alter the detail level of the fire simulation. As the visualization and simulation components of the fire rendering are fairly independent, the LOD methods used are also independent. Thus, different combinations of methods may be used.

The idea behind the LOD algorithm is to use techniques presented in Chapter 2 and 3 to implement dynamic LOD for the fire rendering presented in [KESR06]. The way the fire rendering is implemented affects which LOD techniques are most suitable to get the best performance and visual results. Thus, the implementation of the fire rendering needs to be considered when evaluating different LOD methods.

4.2 Fire rendering considerations

When applying an LOD algorithm to the fire rendering, there are certain considerations that need to be made. Certain methods will be more suitable to implementation than others, depending on the methods used for the fire rendering itself. The way the fire rendering is performed needs to be considered when choosing methods for overall LOD calculation, Visualization LOD, and Simulation LOD. Additionally, the fact that the fire rendering runs on the GPU impacts the way calculations are performed and results stored.

4.2.1 Overall LOD

The overall LOD is a combined evaluation of different LOD conditions that determines the detail level the fire will be rendered with. The fire rendering itself places no restrictions on what LOD conditions are most suitable for use. The overall LOD calculation is fairly independent of the fire rendering itself. Overall LOD is a value declaring what detail level the fire, or any other object, should be rendered with to get the best result in terms of performance and visual quality. The overall LOD value thus depends more on the nature of the application surrounding the fire rendering than the fire rendering itself, meaning the fire rendering implementation does not affect what LOD conditions are most suitable for overall LOD calculation.

4.2.2 Visualization

[KESR06] presents three different ways to perform the visualization of the fire: Black-body radiation tables, a system of textured particles, and volume rendering by ray marching. Black-body radiation gave excellent results for a two-dimensional fire rendering, but proved difficult to extend to three dimensions. As the volume rendering method gave relatively unconvincing results, the system of textured particles will be used here.

The particle visualization works by having a set of particles that follow the motion of the fire fluid. The color and transparency values of a particle are given by the properties of the fluid at a particle's current position. Eventually a particle will pass beyond the core of the fire, or its visibility will be reduced as it cools down. When the visibility falls below a certain threshold, the fire respawns at the base of the fire. The performance of the particle visualization is primarily dependent on the number and size of the particles. Thus, to improve the efficiency of the visualization, adjusting the particle count and particle size are probably the best options.

4.2.3 Simulation

The physical simulation for Rødal and Storli's fire rendering is done by performing CFD calculations on a grid of discrete cells. Each frame the values of the grid are recalculated to emulate the effect of a flowing fluid. The computational complexity of these calculations is high, so the resolution of the grid is an important factor in performance. To increase performance of the simulation, it is necessary to either find ways to dynamically alter the size of the grid or simplify the calculations used.

4.2.4 GPU considerations

As the data for the fire rendering is stored in texture memory there are certain considerations that need to be made when constructing the LOD algorithm. As transferring the data to the CPU, performing calculations, and then transferring back would introduce significant delays, it is preferable to perform the LOD calculations on the GPU. The result should also be stored directly on the GPU without being passed through the CPU.

Performing LOD calculations

The LOD calculations should preferably be calculated on the GPU. Calculations are performed on the GPU by using programmable shaders. A piece of geometry (most commonly a single square) is sent to the GPU. Programmable shaders then calculate the color values for each pixel using lighting calculations and texture information. By putting input data in textures the shader can then be used to calculate most any problem. The physically based fire rendering uses this method to calculate the fluid properties. The LOD algorithm should run on the GPU where doing so is reasonable. It is reasonable that any part of the algorithm that alters data stored on the GPU also runs on the GPU.

Data storage and retrieval

The result of shader calculations is commonly stored in a color buffer which is then rendered to the screen. It is possible to read pixel values from this buffer back to the CPU, but doing so is slow and inefficient as the data transfer speed is far lower between GPU and CPU than internally on the GPU. It is possible to avoid this inefficiency by rendering the shader result directly to texture memory, as is done for the fire rendering algorithm. The LOD algorithm should store its result directly in texture memory to avoid the inefficiency of transferring large amounts of data between GPU and CPU.

4.3 Overall LOD calculation

The overall LOD value of a fire rendering is an indication of the detail level the fire will be rendered with. The value used to alter the detail level of the fire rendering is calculated based on a set of conditions. Additionally, view culling may be used to replace the complex fire simulation with a simpler simulation if the fire is outside the current view.

4.3.1 LOD conditions

The overall LOD value of the fire is calculated by evaluating certain conditions. These conditions are chosen from the list presented in Chapter 3.1. The suitability of each condition will be considered, and one or more will be chosen for implementation and testing.

- **Distance from viewpoint:** Evaluates the distance between the fire object and the camera. Distance is simple to implement and fast to calculate, but may be unsuitable if the camera's field of view is subject to change. Field of view changes might for instance be used to zoom the view, so that an object that is actually far away fills the screen. In such a case, distance would be a poor condition, since the object that fills the screen would still be rendered using a low detail level.

- **Size in pixels:** Somewhat similar to evaluating the distance between fire and camera, but pixel size avoids the effects of changing camera field of view by only evaluating how large the object is in the view. Size is more complex to calculate, however, and changing field of view is not likely to be a factor for the fire application.

- **Object speed:** Calculates the LOD value by evaluating the fire object's speed across the view, meaning its speed relative to the camera's rotation. This is an attempt to take advantage of the fact that it is harder to discern detail on objects moving quickly. Calculating the relative speed is simple and fast, but there is a risk of noticeable popping if an object's relative speed is constantly changing.

- **Obscuring objects:** Attempts to determine whether or not the fire object is obscured by other objects in the scene. If the fire is partially or completely concealed by another object, the LOD value may be lowered accordingly. Complex to calculate, but may offer a good increase in efficiency.

Some of the conditions listed in Chapter 3.1 are concerned with how the characteristics of human vision and mentality affect how we interpret a view, and where we are likely to direct our attention. These conditions will not be considered further as the simple application surrounding the fire will not attempt to model any of these effects. The result of evaluating the chosen conditions will be used to alter the detail level of the fire rendering. While performing the overall LOD calculation on the GPU should be possible, it may be better to do so on the CPU. The reason for this is primarily that the particle system still needs to pass all the particles from the CPU to GPU. Since the overall LOD value may be used to alter the number of particles, it should therefore be calculated and stored on the CPU.

If the LOD framework is later to be implemented for a fire which uses geometry shaders to render completely on the GPU, however, the calculation of the overall LOD value may also be moved to the GPU, as information about the application's camera will be available to shader programs.

4.3.2 View culling

When an object that is subject to physical simulation is no longer visible on the screen, the simulation of the object can often be simplified considerably, depending on the nature of the object in question. The important thing about the simplified simulation is that it maintains the illusion that the object is still undergoing full simulation. An obvious example of this is an object undergoing a predictable motion. If the view pans away from the object for some time and then back, and the object is still in the same position, a viewer will easily realize that the simulation has been halted in between. This ties in with the three problems of consistency, completeness and causality presented in Chapter 3.2.2. However, if the position of the object is approximated in some way as the view pans back, it may be possible to make it appear to the viewer as if the object has been simulated constantly, while still drastically reducing the complexity of the simulation.

For a fire rendering, a possible solution is to simply halt the rendering of the fire altogether when the view pans away from it. The justification for doing this lies with the chaotic nature of a physically based fire rendering. Unlike with an object undergoing easily predictable motion, it is very hard to accurately predict how the fire should look in the future. Thus, a viewer will probably not be able to tell the difference between a rendering that has been running constantly for a set amount of time, and a rendering that has been halted for a portion of that time. This effect can be seen in Figure 4.1. View culling may be done by checking if the fire is currently within the camera's field of view. One aspect that may need special consideration, however, is the inclusion of dynamic lighting. Even if a fire simulation is halted due to view culling, the fire should still give off light to its surroundings, preferably with the same flickering effect gained from the full simulation. For this, a simple proxy simulation may be used for the lights in order to get the desired effect.

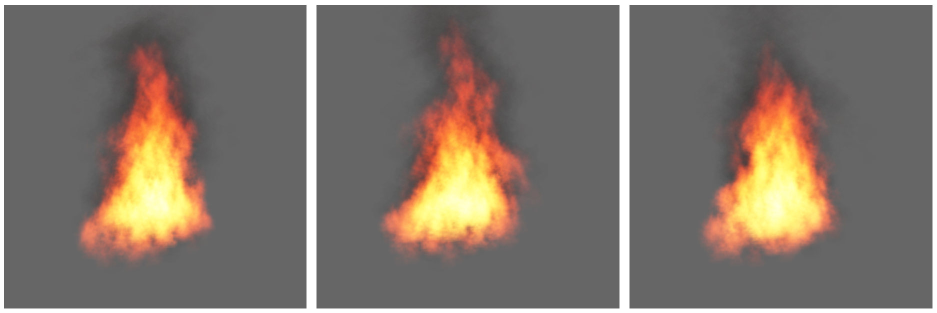

Figure 4.1: Illustrating the difficulty in predicting future states of a fire rendering. Left: Fire state at starting point. Middle: Fire state after halting for some time and only running a few frames. Right: Fire state after running continually for some time.

4.4 Visualization LOD

The main component of the fire visualization is the particle system. The most important aspect of the visualization LOD is therefore to change the particle system to increase efficiency as the fire becomes less important to the view. Additionally, changes to the dynamic lighting system may be used to maintain the flickering effect of the fire while the fire itself is not rendered.

4.4.1 Particle clustering

The clustering method presented in [DO01] works by grouping particles that share certain characteristics such as position and velocity. Each particle in the group is visualized individually, but the entire group behaves as a single particle physically. However, as the particles in the fire rendering simply fetch their simulation data from the fluid simulation, there is little performance to be gained from this approach. Additionally, the information about other particles needed to evaluate possible clusters would be hard to access on the GPU, because when sent to the GPU, each particle is treated independently, without access to the data of any other particles. This means that this method would probably have to be implemented on the CPU, or large changes would have to be made to the way particles are stored and rendered.

4.4.2 Altering particle count and size

With Rødal and Storli's implementation, the size and number of particles for the fire rendering are constant. Particles that pass beyond the fire respawn at the base. When the fire is small or otherwise unimportant for the view, the particle system will usually contain a lot of redundancy in that several particles will occupy the same space on the screen. Removing some of the redundant particles would increase performance. As such, lowering the particle count may be a good way to increase the efficiency of an unimportant fire object.

A possible way of altering the particle count is to merge particles together. Several particles are replaced by a single, larger particle. While this may give a fairly good performance increase, the method has a major drawback similar to particle clustering. As mentioned earlier, on the GPU each particle has no information about the other particles. As for particle clustering, particle merging requires the ability to determine which particles share similar attributes such as position and speed. Having this information on the GPU would require changes to the data structure, and would also be costly to calculate. For these reasons, particle merging would need to be implemented on the CPU.

Another possibility is to use the respawning functionality of the particles to alter the particle count. However, stopping all respawning until the particle count reaches the desired level might create unnatural gaps in the fire. Instead, the ratio between current and target particle counts may be used to control respawning. If the target particle count is half that of the current particle count, only every other particle would respawn, thus gradually lowering the particle count to the desired level. Reducing the particle count might produce gaps in the fire, which may be counteracted by increasing the size of each individual particle. Additionally, the apparent overall intensity of the fire is created by the contribution of many partially transparent particles. To prevent the intensity from changing when the particle count changes, the intensity of each particle should be changed according to the particle count.

Currently the number of particles used by the fire rendering is governed by the CPU, as each particle needs to be passed from the CPU to the GPU for rendering. The change in particle count and respawn ratio would also need to be implemented on the CPU, which could potentially slow down the rendering since a message needs to be passed from the GPU to CPU when a particle decays. With a geometry shader the particles are generated on the

GPU, and so the respawning algorithm could also be moved to the GPU, resulting in higher performance. The size of particles can be controlled by the GPU by passing a single parameter to the geometry or vertex shaders.

4.4.3 Altering particle textures

As the importance of a fire object is reduced, the detail of the fire particles becomes less important. When the overall detail level is low, reducing the detail level of the particle textures may increase performance without hurting the fire's appearance noticeably. On newer GPU chipsets it is unlikely that resizing the particle texture would have a large impact on performance, but disabling the texture may have a larger impact. When the overall LOD value is low enough, the textures can be disabled and the particle rendered using a single color.

4.4.4 Altering the dynamic lighting system

The dynamic lighting system presented in [KESR06] works by having a number of stationary or moving lights within the fire. Each light's intensity is set according to the properties of the fire fluid at the light's position. The changing properties of the fluid and thereby of the lights create the characteristic flickering effect of the fire. Any changes to the lighting system should preserve this flickering effect.

To reduce the detail level of the lighting system, it may be possible to reduce the number of lights used while increasing the intensity of each light. Additionally, the motion of each light may be changed to a simple, predefined motion instead of following the fluid motion. It should be possible to implement changes to the lighting system on the GPU, simply using different shader code when employing a simpler version of the lighting simulation.

4.4.5 View culling

While the effect of changing the parameters of the visualization in [KESR06] is very dependent on the simulation detail level, disabling the visualization completely should have a significant impact. This is especially true if the visualization is being also disabled, or rendered at a low detail level.

When the fire is outside the current view the majority of the visualization may simply be switched off, as the results would not be visible. However, the dynamic lighting system affects the scene in general, and therefore requires consideration.

If the simulation is running normally when the visualization is culled, the lights may be updated as normally. If the simulation also undergoes some form of view culling, however, the lights would become static, and the characteristic flickering effect would be lost. To avoid this, the lighting system may use a simple proxy simulation instead of evaluating the fluid properties. A stochastic calculation may be used to control the motion and intensity of the lights, preserving the lighting's flickering effect.

4.5 Simulation LOD

The simulation of the fire rendering is governed by CFD calculations performed on a grid of discrete cells. The complexity of these calculations and the resolution of the grid has a significant impact on performance. Adjusting the grid resolution or simplifying the calculations used is therefore a good way to increase performance.

4.5.1 Altering CFD grid resolution

The resolution of the grids used for CFD calculations has a large impact on the performance of the application. Thus, dynamic resizing of the CFD grids is an important aspect of increasing the efficiency of the fire rendering.

One way to resize the grids is to simply delete or add columns, rows and slices along the edges until the desired size is reached. This is likely to be fairly simple to do, but significant amounts of data might be lost when reducing size. The method may also lead to odd appearance when increasing size as the fire would ”grow” into the added cells.

Another possibility is to use methods for the resizing of polygonal meshes, as presented in Chapter 3.5.2. There are numerous different such methods, but they rely on the non-uniformity of most polygonal meshes to alter their structure. As such, mesh resizing methods are not very suitable for the uniform CFD grids. Additionally, implementing such methods on the GPU would be complicated, as seen in [JJ05] and [FR06].

A third option is to adapt methods for image filtering, presented in Chapter 3.5.1. There are many different image interpolation techniques, but only nearest neighbor, bilinear and bicubic filters are considered here. These three methods give the cells in the new grid values based on nearby cells in the old grids. While the methods usually work on two-dimensional grids, it is fairly simple to extend them to a three-dimensional grid.

Nearest neighbor filtering simply selects the value of the cell in the old grids that is closest to the cell in the new grid. Nearest neighbor is fast and simple, but data may be lost as some cells are discarded completely when the grid resolution is lowered. Bilinear filtering normally works by giving the new cell a value based on the 2x2 closest cells in the old grid. For the three-dimensional grid the value would be taken from the 2x2x2 closest cells. Bilinear filtering is more accurate than nearest neighbor, but also slower. Finally, bicubic filtering would use the 4x4x4 closest cells to determine the value of the new cell, again increasing the accuracy and cost of the method.

As the CFD grids are stored on the GPU as texture slices, it should be possible to implement the resizing method on the GPU by rendering to texture, similarly to how the fire rendering itself is performed. As the act of sampling values from a texture is a relatively costly function, it may be desirable to minimize the number of texture lookups used. This is particularly true when the samples will be done from different areas of the texture, as would be the case for texture slices. Additionally, consideration needs to be given to whether to generate new textures of appropriate sizes when resizing, or simply changing which parts of the textures are used. The former is slower, but helps conserve texture memory, while the latter is faster but does not free up texture memory as the detail level is lowered.

4.5.2 Simplification of simulation calculations

The fire rendering method presented in [KESR06] performs full simulation steps for each time step of the rendering. When the fire is not important to the current view it may be possible to reduce the complexity of these calculations. Two ways to do this could be to use a simple proxy simulation for the fluid behavior, or to use simulation step skipping.

A proxy simulation could be simple calculations that alter the motion of the fire fluid according to known models of fire behavior, with the addition of simple stochastic variations to prevent the fire from looking like a static animation. Using a proxy simulation should reduce the cost the the simulation, though the fire will likely look unconvincing if examined by the viewer.

Simulation step skipping means not performing the fluid simulation for every frame that is rendered. If the simulation calculations are halted, the fire particles would still travel through the system based on the current fluid state, though the appearance would be that of a static animation. By not performing the CFD calculations for each time step, performance could be increased without hurting the appearance of an unimportant fire object. It is possible that simulation step skipping could adversely impact the appearance of the fire motion, but testing should reveal any such issues. Using proxy simulations on the GPU could be done by invoking different shader code for the simulation, while step skipping would be done simply by not invoking the simulation of the fire every frame.

4.5.3 View culling

As mentioned in Chapter 4.3.2, predicting future states of the fire simulation is very difficult. Thus, a possibility is to simply halt simulation while the fire is outside the view. Doing so should significantly increase the speed of the application without a viewer being able to notice a difference in appearance. However, as mentioned before, the dynamic lighting calculations for the fire rendering would need to be replaced by a proxy simulation to preserve the appearance of the light the fire gives off to its surroundings.

4.6 Conclusion

Of the different methods presented, some are more suitable for use with the type of fire rendering considered as a basis. Based on their perceived suitability, some of the methods are chosen for inclusion in the LOD framework.

4.6.1 Overall LOD

What conditions are most suited for calculating the overall LOD value of an object will depend on the nature of the application used. The conditions are chosen based on the expected benefit when running with Rødal and Storli's fire rendering as well as how general the condition is. Conditions that depend on the nature of the application the fire is used in are not included for the framework, though they may give good results for other applications.

To calculate the overall LOD value of a fire object, a simple distance evaluation will be used. While distance is not suitable for every application, it is still the most general in that it is simple to calculate and works well in most situations. The other conditions are more complex to calculate and not as generally suitable. One method that may also give a good result in general is to check where any other objects obscure the view of the fire. However, the best way to evaluate this condition is most likely to use the z-buffer on the GPU. This makes checking for obscuring objects more suitable for a fire rendering where the particle system also runs complete on the GPU, using a geometry shader. Thus, for the current framework only distance will be used to calculate the overall LOD value.

The overall LOD value will be at maximum when the fire is within a certain distance from the camera. It will then be lowered gradually as the fire moves away, until it reaches a minimum level. The formula used to calculate the distance LOD is seen in Equation 4.1.

$$lod = min(\frac{b}{d}, 1.0) \tag{4.1}$$

lod is the overall LOD level, b is a base value determining how far away the fire will be when the detail level starts to decrease, and d is the distance between the fire object and the camera. e

Additionally, view culling will be used when the fire is outside the current view. The application will check if the fire is visible or not and store the result separately from the overall LOD value. Because of the chaotic nature of the fire and the difficulty in predicting its movements while outside the view, view culling is used to halt certain aspects of the rendering and replace others with simple proxy simulations. The angle between the camera and the fire will be compared to the field of view of the camera to determine if the fire is visible or not. Figure 4.2 shows how the visibility and distance evaluations will be used by the application to determine how the fire is rendered.

4.6.2 Visualization

The overall LOD value for the fire will be used to alter the parameters of the particle system used for visualization. When the overall LOD value is lowered, the number of particles will be reduced. Because of the way the particle system in [KESR06] works, and because it is preferable to implement the LOD algorithm on the GPU where possible, the respawn functionality will be used to reduce or increase the number of particles. When the overall LOD value is changed, a respawn ration will be calculated by Equation 4.2, where r is the respawn ration, tc is the desired particle count, and cc is the current

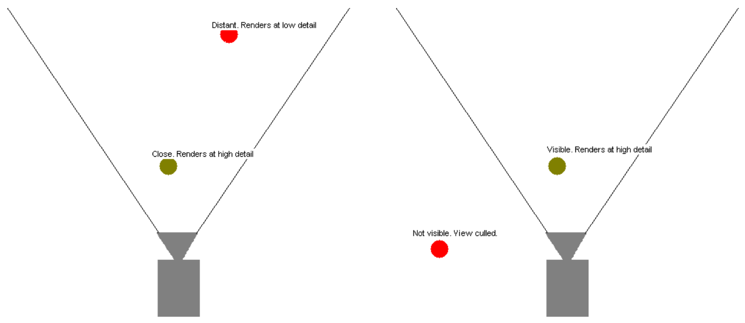
Distant. Renders at low detail
Close. Renders at high detail
Visible. Renders at high detail
Not visible. View culled.

Figure 4.2: Illustration of how the overall LOD is calculated. Left: Distance LOD. Right: View culling.

particle count. The target particle count will be calculated using Equation 4.3, where mc is the maximum particle count, and lod is the overall LOD value of the fire. To balance the effect of changing the number of particles, the size and intensity of each particle will change as well. Size will be changed according to Equation 4.4, where s is the calculated particle size, and ms is the particle size at the highest detail level. The size is not changed linearly, as that would make the particles too large too quickly. Particle intensity will be calculated using Equation 4.5, where i is the calculated intensity and mi is the intensity at the highest detail level.

$$r = \frac{tc}{cc} \tag{4.2}$$

$$tc = mc * lod \tag{4.3}$$

$$s = \frac{ms}{\sqrt{lod}} \tag{4.4}$$

$$i = \frac{mi}{lod} \tag{4.5}$$

For the particle textures, the same texture will be used for particles of different sizes, giving different relative texture resolutions. Additionally, at the lowest detail level the particles will be rendered without textures. The dynamic lighting system will be replaced by a simple proxy simulation when the detail level drops below a certain threshold. The proxy simulation will use simple stochastic calculations to vary the position and intensity of the lights.

When the fire object is not visible in the current view, view culling will be used and the visualization of the particle system will be turned off. No aspects of the visualization will be calculated, except for the dynamic lights, which will be calculated using the simple proxy simulation.

4.6.3 Simulation

The most important component of the simulation LOD method is the dynamic resizing of the CFD grids, which are stored as textures on the GPU. To resize the textures, nearest neighbor filtering will be used. Because the LOD method will be continuous, each change to the grid size should be small, thus reducing much of the risk of losing data. The chaotic nature of the fire should also make any such data loss difficult to detect. Additionally, as texture sampling is a relatively costly operation on the GPU, it is preferable to have as few texture samplings as possible. The grid size will be changed according to Equation 4.6, where s is the calculated grid size, and mr is the grid size at the maximum detail level.

$$s = mr * lod \tag{4.6}$$

An illustration of how the grid resizing occurs between a 3x3-grid and a 15x15-grid can be seen in Figure 4.3. Errors that may occur are displayed in the Figure, but exaggerated, as the grid resizing will be continuous, thereby taking making smaller steps at a time.

Simulation step skipping will be used to reduce the number of CFD computations performed when the overall LOD value is lowered. The number of frames between each new CFD computation will be determined by Equation 4.7, with t being rounded to the nearest whole number.

$$t = \frac{1}{lod} \tag{4.7}$$

Finally, as for the visualization, the simulation will be halted when the fire is outside the current view, due to the difficulty in spotting such pauses as mentioned in Chapter 4.3.2. One exception is in a fire that has just been ignited. A recently ignited flame should be simulated fully for a certain period of time, until it has gone beyond the initial ignition phase.

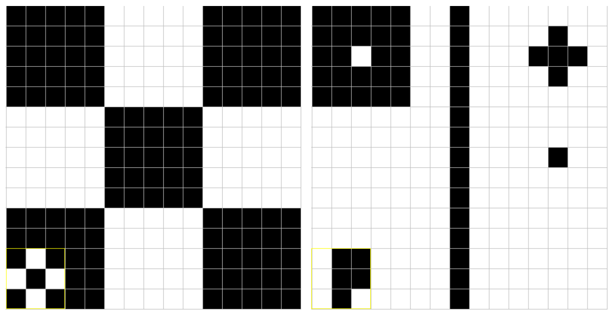

Figure 4.3: Grid resizing using nearest neighbor. Framed area in bottom left is the 3x3-version of the grid. Left: Grid scaled up. Right: Grid scaled down.

4.7 Summary

A quick summary of the methods chosen to be part of the framework for including LOD with a physically based fire rendering is outlined below.

- **Overall LOD:** The overall LOD value of a fire object will be calculated using a simple distance measurement between the camera and the fire.
- **View culling:** A view culling parameter will be set to either on or off depending on whether or not the fire is visible in the current view.
- **Visualization:** As the overall LOD is changed, the number of particles and each particle's size and intensity will change accordingly. The number of particles will be changed by varying how many particles respawn when a particle leaves the fire. The same particle texture will be used for all particle sizes, and at the lowest detail level particles will be rendered without textures. When the fire is not visible, the visualization will be turned off, except for the dynamic lighting system, which will be replaced by a simple proxy simulation.
- **Simulation:** Nearest neighbor filtering for image resizing will be used to resize the CFD grids, stored in textures, as the detail level changes. As the overall LOD value lowers the CFD computations will no longer be performed each frame, with the number of steps skipped increasing as the detail level lowers further. Finally, the simulation is halted completely when the fire is not visible.

As the methods are tested and evaluated, changes may be made to improve the quality of the algorithm.

Chapter 5

Implementation

The implementation of the framework presented in Chapter 4.6 could not be completed in time for the thesis deadline (the reasons for this are discussed in Chapter 7.1.) Thus, this chapter describes a partial implementation of the framework, becoming more a plan for an implementation of most of the intended features of the framework.

The implementation presented in this chapter consists of the overall structure of the application, the implementation of the fire rendering itself, and the implementation of the different aspects of the LOD algorithm.

The application structure describes the overall construction of the entire application, governing scene initialization, object relationships, and input handling. The fire rendering structure describes the implementation of the physically based fire rendering, which is based on Rødal and Storli's fire rendering. The LOD algorithm describes how the application calculates overall LOD values for different objects and how these values are used to alter the detail level of each fire object in the scene.

5.1 Overall structure

The framework is implemented using Simple DirectMedia Layer[1] for graphics initialization and CG[2] for GPU shader programs.

The main components of the framework are the classes Scenegraph, Camera, Node, FireObject, FireSimulator, ParticleSystem, and FireParameters.

[1]http://www.libsdl.org

[2]http://developer.nvidia.com/object/cg_toolkit.html

The class ModelObject is also included as an example of an object in the scene that is not a fire. SceneObject is the abstract class from which ModelObject and FireObject inherits. The functionality and relationship between the classes is outlined below.

- **Camera:** The camera class contains the data for the current viewpoint in the scene. The class has methods that may be used to move and rotate the view according to user input or other parameters.

- **FireObject:** Implements the abstract SceneObject class. The FireObject class functions as a wrapper for the different components of the fire rendering, passing on calls for LOD calculations or other parameter changes and initializing the fire rendering according to a specified configuration file.

- **FireParameters:** Stores all the parameters for a fire object as well as references to the different textures and shader programs used for rendering and LOD operations. The FireParameters class serves as a centralized storage for the fire object's parameters to make changing those parameters easier for the LOD operations.

- **FireSimulator:** Contains functionality for the fluid simulation portion of the fire rendering, as well as LOD operations to be performed on the fluid simulation. Stores references to the 3d textures used for rendering, the central FireParameters instance, and the GPU shader programs used to perform simulation steps and LOD operations on the CFD grids.

- **ModelObject:** Implements the abstract SceneObject class. Stores a single geometry-object loaded from file that may be rendered in the scene together with fire objects or other model objects.

- **Node:** Stores a reference to a single object in the scene. The object may be any type that implements the abstract SceneObject class. Node contains functionality for calculating its overall LOD value and passing it on to the SceneObject it governs. A Node may also have a series of child nodes, which are normal Nodes that inherit position and other parameters from their parent.

- **ParticleSystem:** Contains functionality for the visualization of the fire rendering, as well as LOD operations to be performed on the visualization. Stores references to the 3d textures used for the visualization of the fire, the central FireParameters instance, and the GPU shader programs used to render the system of textured particles and perform LOD operations on the particle system.

- **Scenegraph:** Contains all data for the current scene. Stores references to the current Camera instance used and a list of Node-instances. Contains functionality for adding nodes to the scene, updating all nodes in the scene, rendering all nodes in the scene, and handling user input.

- **SceneObject:** Abstract class. Specifies the functionality required for objects in the scene, as well as common data. Contains virtual functions for updating, rendering and LOD calculation.

- **Texture3d:** Stores data for the grid of discrete cells used by the fire rendering. Emulates a 3d textures by using slices. Each slice is stored as a 2d texture with the specified width and height, with the number of slices being the depth of the 3d texture. Because the same texture may not be used as both input and output for a GPU shader program, the result of a computation is written to a temporary texture. The temporary texture is then swapped in so that the old input texture becomes the new output texture and the old output texture becomes the new input texture. Texture3d also contains functionality for resizing.

The framework also contains peripheral functionality for initialization, such as initializing OpenGL and CG, and loading objects from files. The overall hierarchy of the application is illustrated in Figure 5.1.

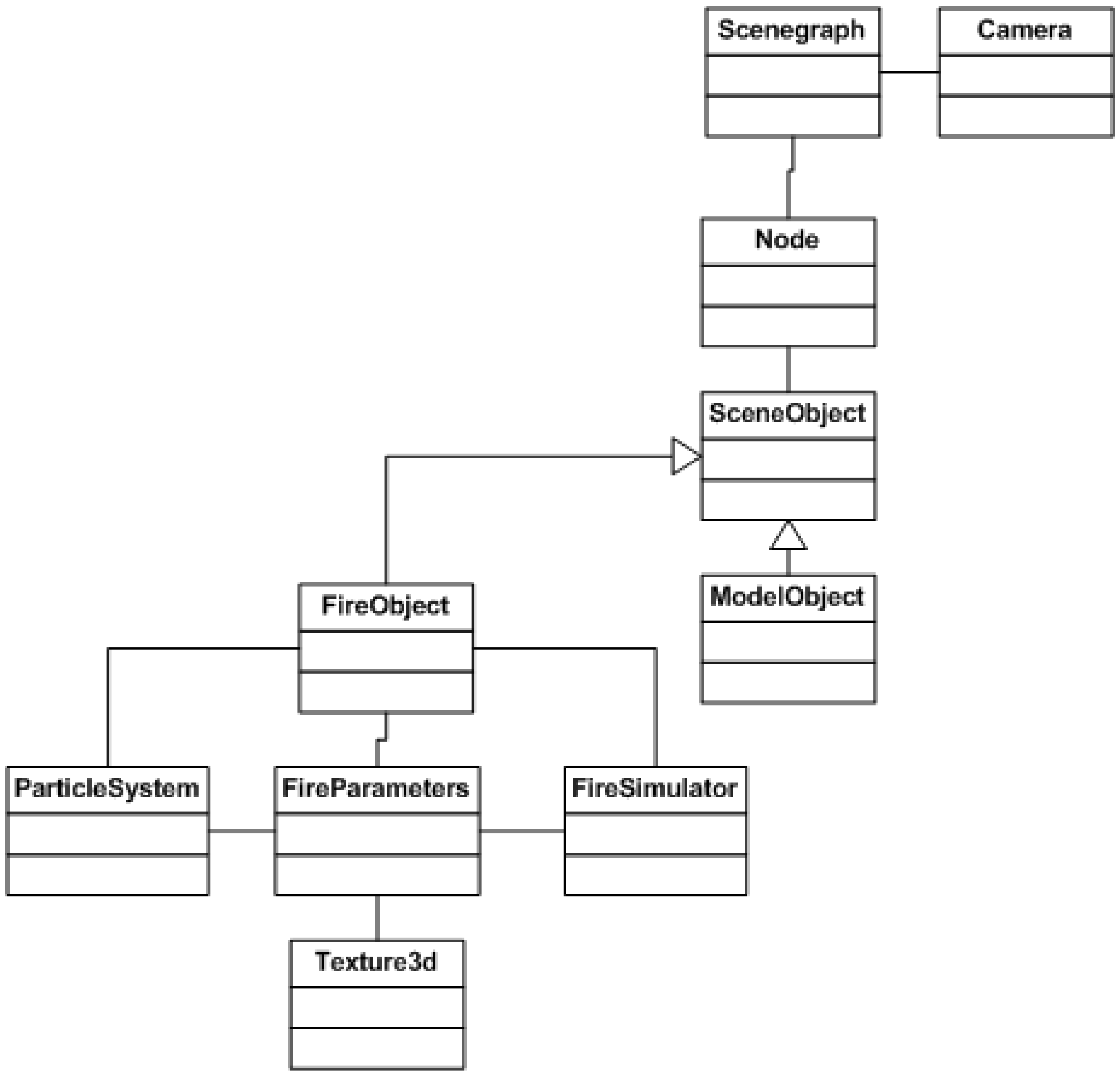

Figure 5.1: Application hierarchy.

5.2 Program flow

For each frame of the application a number of operations are performed by the application main loop. These operations are divided into the general functionality of the application for visualization and simulation of scene objects as well as input handling, and LOD operations. The LOD operations consist of the different operations performed to calculate overall LOD values, fire visibility, and visualization and simulation LOD.

5.2.1 General functionality

The general functionality of the application consists of a rough description of all the steps performed by the main loop for each frame rendered. These steps are outlined below and illustrated in Figure 5.2.

1. **Update:** The list of nodes in the scenegraph is traversed, and the update()-method run for each. What update() does depends on the type of object stored in the node. For a fire object, update() will run a single time step for the fire simulator, calculating new values for the fire fluid textures.

2. **Calculate LOD:** Performs LOD operations for each node in the scenegraph. The type of operation will depends on the type of object. Each node will calculate an overall LOD value and pass it on to the SceneObject it has stored. The details of LOD operations for a fire object are specified further down.

3. **Render:** The list of nodes in the scenegraph is traversed, and the render()-method run for each. Render() runs the visualization for the node's object, typically displaying it on the screen. For a fire object, render() will perform a visualization step by rendering all the particles in the particle system according to the fluid values at each particle's position.

4. **Handle input:** Handles any input from the user such as key presses and mouse movements, typically to alter the motion of camera or objects or other parameters for the scene.

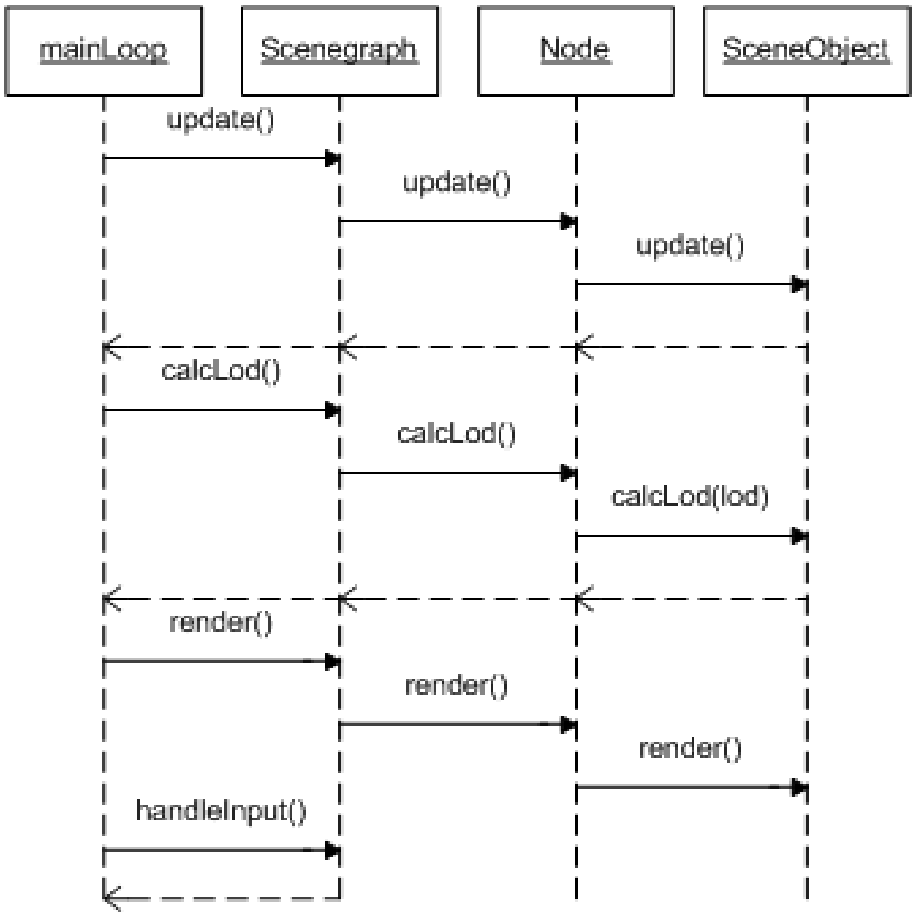

Figure 5.2: General functionality of the application.

5.2.2 LOD operations

The LOD operations performed on the fire objects in the scene are outlined below, with more detailed looks at Visualization and Simulation LOD given further down. The sequence of steps for these operations are also illustrated in Figure 5.3.

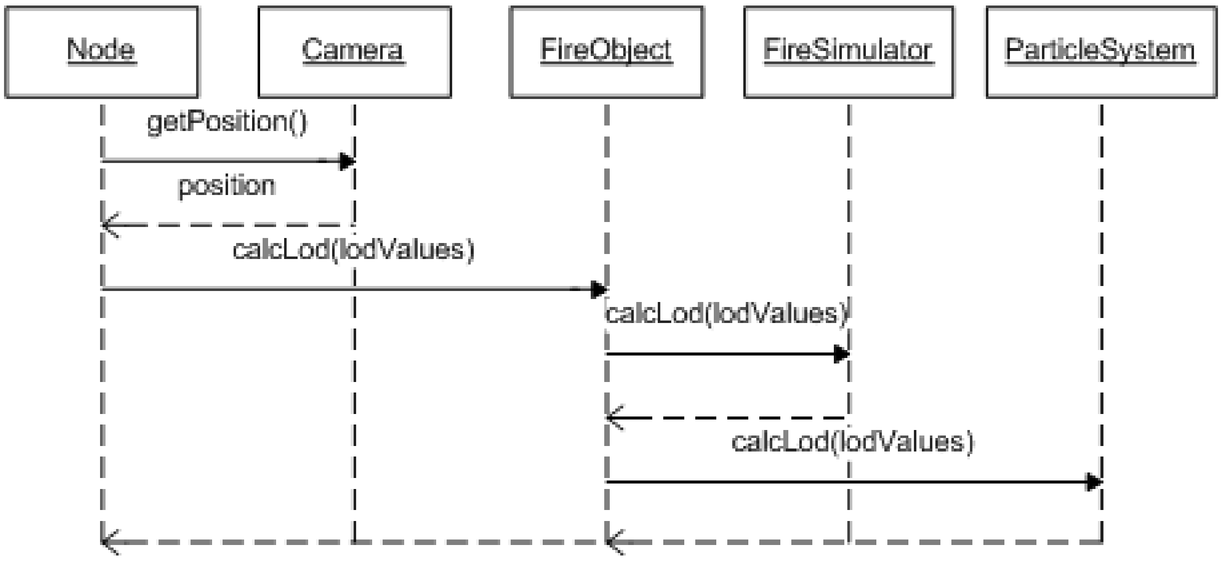

Figure 5.3: LOD operations on fire objects.

1. Each node's calcLod()-method is invoked from the scenegraph with a reference to the current camera.

2. The node fetches the camera's current position using getPosition().

3. The node calculates the distance between itself and the camera, and uses the distance to calculate the overall LOD value.

4. The node determines whether its object is visible or not by calculating the angle between it and the camera's view direction, and comparing the result to the camera's field of view.

5. The node invokes the calcLod() method of its object, passing along the overall LOD value and the visibility rating.

6. The FireObject instance uses the visibility rating to determine whether the visualization and simulation should be switched off, and a proxy simulation used for the dynamic lighting. The simulation must be run for a certain period after ignition so the fire has progressed to a stable burning state.

7. If the overall LOD value has changed, it is then passed on to the FireSimulator and ParticleSystem's calcLod()-methods. If the overall LOD value has not changed, no further steps are performed.

8. The FireSimulator uses the overall LOD value to alter the detail level of the fire simulation by setting simulation step skipping parameters and resizing the CFD grids. A more detailed look at this step can be seen in Chapter 5.2.3.

9. The ParticleSystem uses the overall LOD value to alter the detail level of the fire visualization. A more detailed look at this step can be seen in Chapter 5.2.4.

5.2.3 Simulation LOD

The simulation LOD consists of two main parts: simulation step skipping and CFD grid resizing. Pseudo code for the simulation LOD can be seen in Figure 5.4. The step skipping is simply an integer telling the application how many frames should be rendered between each time a simulation step is performed. When the fire simulator is invoked, it compares a counter to this value. If the counter is below the value, the counter is incremented and the simulation step is skipped. If the counter has reached the threshold value, the counter is reset and the simulation step is performed.

The CFD grid resizing is done using a form of nearest neighbor filtering. The 3d textures are stored as a set of 2d texture slices, with the number of slices being the depth of the texture. When the 3d textures are resized, the textures themselves are not resized. What changes is simply how much of the texture is used. This is done to avoid the likely slowdown of constantly having to generate new textures. The textures are scaled according to the ratio between the new and current grid resolutions. Depth scaling is done by going through each slice in the new grid, and using the nearest slice from the old grid as the input texture. The individual texture slices are scaled by altering the texture coordinates used to render the slice. By scaling the texture coordinates according to the ratio between new and old grid resolutions, the values of the slice are stretched or compressed to a larger or smaller area of the texture.

```
/**
 * FireSimulator's calcLod()-method
 * @param lod is a value between 0 and 1
 */
void calcLod(float lod){
  // Time step skipping
  stepSkip = (int)1/lod;

  // CFD grid resizing
  for each 3dtexture t used by fire rendering{
    t->resize(lod);
  }
}

/**
 * 3d texture's resizing method. Size
 * is a simplification of the values width,
 * height and depth.
 * @param lod is a value between 0 and 1
 */
void resize(float lod){
  newSize = maxSize * lod;
  scaleFactor = newSize/currentSize;

  for i from 0 to newSize{
    nearestSlice = i * scale;
    bind input texture nearestSlice to input;
    bind output texture i to output;
    texCoords = currentSize*scaleFactor;
    render slice to output texture using texCoords;
  }
  swap output and input textures;
  currentSize = newSize;
}
```

Figure 5.4: Pseudo code for simulation LOD.

5.2.4 Visualization LOD

The visualization LOD consists of the changes made to the particle system. These changes include altering the count, size, and intensity of particles. Pseudo code for the visualization LOD can be seen in Figure 5.5. The particle count is changed linearly with the overall LOD value. The count is changed by altering how particles spawn. Normally, as a particle's visibility passes below a certain threshold it decays (is removed), and is respawned at the base of the fire. The LOD algorithm changes this by setting a spawn ratio so that more or less than one particle may spawn each time a particle decays. The spawn ratio is the ratio between the current particle count and the target particle count.

As the particle count changes, the size and intensity of each particle also changes. The intensity and size of each particle changes with the overall LOD value to compensate for having fewer or more particles. The changed parameters are only applied to new particles. Additionally, if the overall LOD value is at a minimum level, particle textures are disabled.

The decayParticle()-method is not a part of the general LOD algorithm, but is changed to incorporate the spawn ratio. When a particle decays a counter is incremented by the current spawn ratio. If the counter's value is higher than 1 a single particle is spawned and the counter is decremented by 1, until its value falls below 1.

```
/**
 * ParticleSystem's calcLod()-method
 * @param lod is a value between 0 and 1

void calcLod(float lod){
  targetCount = maxCount * lod;
  spawnRatio = targetCount / currentCount;

  pSize = baseSize / sqrt(lod);
  pIntensity = baseIntensity / lod;

  if lod is at minimum level{
    disable particle textures;
  }
}

/**
 * Effect of adding spawn ratio to particle
 * respawning
 */
void decayParticle(){
  counter = counter + spawnRatio;
  while(counter >= 1.0){
    spawn particle at random location near fire base;
    counter--;
  }
}
```

Figure 5.5: Pseudo code for visualization LOD.

Chapter 6

Results

This chapter presents the results of various tests run to verify the viability of the LOD framework. Because the full physically based fire rendering with LOD could not be completed in time for the deadline, the results presented here are from tests run on Rødal and Storli's fire rendering. All tests were run on an Intel 1.83GHz Core Duo with 1GB RAM and an NVIDIA Geforce 7600 Go with 512 MB VRAM. For performance tests, the fire would be rendered as shown in Figure 6.1. Tests were performed by manually adjusting the

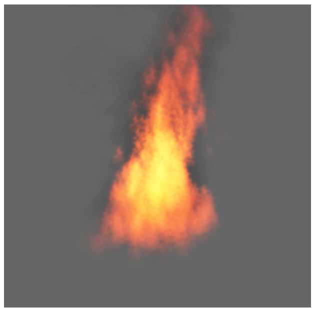

Figure 6.1: Position of fire for performance tests.

detail level of the fire rendering and evaluating the impact on performance and visual quality.

The validity of the results will be considered, as the implementation was not completed in time. Then the different results will be presented, evaluating the cost of running the different LOD operations, and then the results for the different components of the fire rendering. Finally, a brief summary will be given, highlighting the most important discoveries made.

6.1 Validity of results

The results presented are from a series of tests that were run using the incomplete implementation. Most of the tests were run on the application using Rødal and Storli's code as a basis. Because the implementation could not be completed in time, the tests were generally performed by manually changing the detail level of the different aspects of the fire rendering.

While the results of these tests are not conclusive proof of the LOD algorithm's viability, they should offer a good indication of how well the algorithm works and how it will affect the different aspects of the fire rendering.

6.2 Results

The different tests performed will be presented in turn here, and the achieved results will be considered. The tests cover the LOD algorithm, the rendering of multiple fires, variations in the detail level of fire visualization and simulation, view culling, and simulation step skipping.

While the detail level changes made by the LOD algorithm are important for the end result, it is also important to consider the performance impact of the LOD algorithm itself. Performance gains from reducing the detail level of the fire are not much good if the LOD algorithm itself is very costly to run.

6.2.1 Impact of running the LOD algorithm

Tests were run to see how the different aspects of the LOD algorithm such as overall LOD calculation, visualization LOD, and simulation LOD would affect the performance of the application. With one exception, however, none of the LOD calculations made any discernible difference to the application performance. The only exception to this is the grid resizing algorithm. The performance impact of the grid resizing was tested by running the texture resizing on a test texture alongside Rødal and Storli's fire rendering. The number of 3d textures used for testing was the same as the number of 3d textures used in the fire rendering. The resolution of the 3d textures was also the same as that of the ones used for the fire rendering. Each frame, the resize algorithm would run for the full size of the textures (basically resizing them from full size to full size). The results for different grid resolutions can be seen in Table 6.1. As these results show, the negative impact on performance from

Grid size	Grid resizing	No grid resizing
16x24x16	27.82	29.32
24x36x24	13.54	14.19
32x48x32	5.76	5.94

Table 6.1: Performance impact of running the resize algorithm for different grid resolutions.

the grid resizing is very small. Most of the impact that is there is likely to stem from the overhead of rendering slices to framebuffer objects. As such, it may be possible to lessen this small impact by combining the resizing algorithm with the simulation of the fire using an extra parameter, rather than perform the resizing separately. The visual impact of gradually scaling the CFD grids can only be gauged with a functional implementation of the entire method. However, the visual result of gradually scaling a single static texture slice is illustrated in Figure 6.2. Combining the different aspects of the LOD algorithm seemingly produced the exact same result as only running grid resizing, further reinforcing the conclusion that only grid resizing has an impact on performance. The result tables for the combined algorithm and the different portions that had no discernible effect may be seen in Appendix A.

There are a couple of things that may change these results. For the overall LOD calculations, only distance LOD was calculated, which is probably the simplest LOD condition to evaluate. If more complex conditions such as pixel size or obscuring objects are included, the overall LOD calculations may have an impact on performance. Also, when grid resizing is performed, the textures themselves are not resized. The internal values are altered, and the fire rendering changes how much of the texture is used. The amount of texture memory used by a fire rendering is therefore static. Actually resizing the textures may allow for conserving resources, but could also make the algorithm slower.

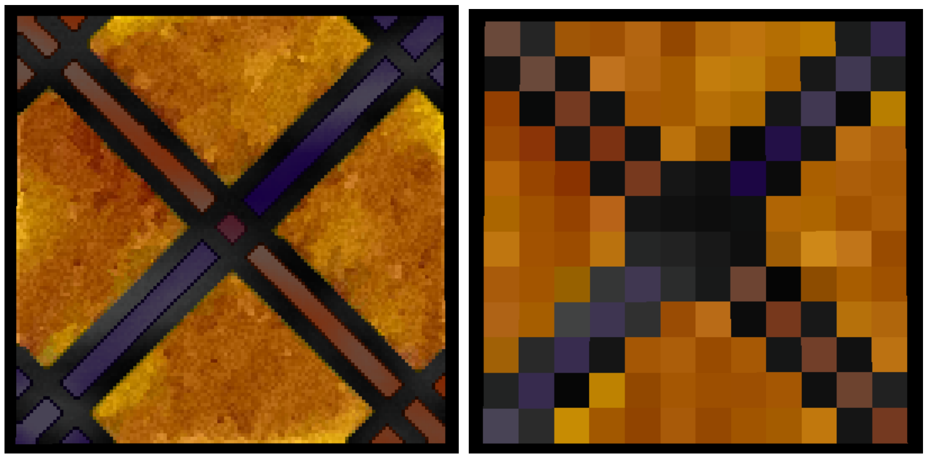

Figure 6.2: Single resized texture slice. Left: Slice is rendered as a full 128x128 texture. Right: Same texture after gradually using nearest neighbor filtering to scale it down to 12x12.

6.2.2 Multiple fire objects

A part of the objective of this thesis is to enable the fire to exist in a large scene with other objects, including other fire objects. Each fire object should have individual simulations, visualizations, and LOD values. Thus, it is important to determine how the performance scales with multiple fire objects. The performance impact of running multiple fire objects at the same time may be seen in Table 6.2 and an illustration of three fires running side by side is shown in Figure 6.3. As is seen, performance scales roughly linearly

No. of fires	Performance
1	30.23
2	14.34
3	9.44
4	7.23

Table 6.2: Performance with multiple fire objects rendered at the same detail level. Grid size 16x24x16, 2048 fire particles and 512 smoke particles. All fire objects are rendered at the same distance from the camera.

with the number of fire objects. This is a promising result, as it indicates that there is not a large amount of overhead involved with including several fire objects.

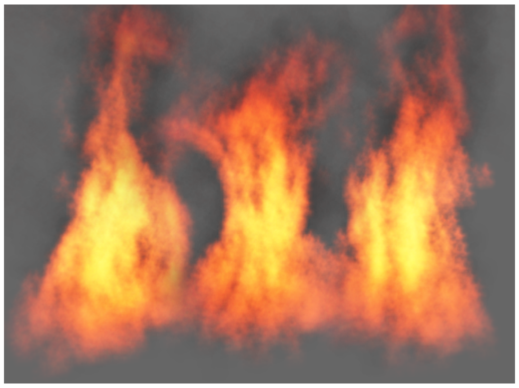

Figure 6.3: Screen shot of three fires running simultaneously.

6.2.3 Visualization and simulation detail levels

The simplest way to alter the detail level of the fire rendering is to alter either the resolution of the CFD grids, or the number of particles used for visualization. Table 6.3 shows the performance from the 3D fire rendering running on the GPU, using different grid dimensions and particle counts. As

	Grid size		
Particle count	16x24x16	24x36x24	32x48x32
512	169.19	32.87	6.58
1024	92.36	22.19	6.12
2048	47.23	15.19	5.94
4096	24.63	11.07	5.54
8192	12.72	7.11	4.87
16384	7.04	5.08	3.93

Table 6.3: Performance results for fire rendering using different grid sizes and particle counts. Fire rendered without smoke.

expected, the impact from altering the particle count is greater when the grid dimensions are small, as the simulation dominates the calculation complexity when the dimensions are larger. This indicates that altering particle count needs to be combined with altering the grid dimensions to be effective. For a grid size of 16x24x16, doubling the number of particles roughly halves the performance of the rendering. For a grid size of 32x48x32, however, even multiplying the particle count by 32 doesn't quite halve the performance.

The visual result of rendering the fire using different resolutions for the CFD grids is illustrated in Figure 6.4. The still images do not fully convey the

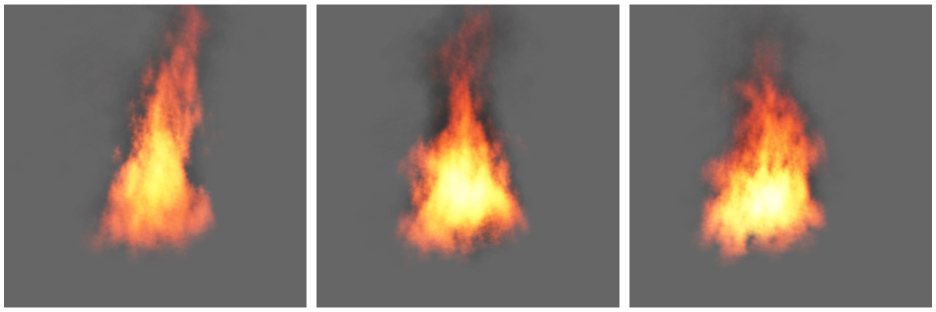

Figure 6.4: Screen shots of fire using different grid sizes and 4096 particles. Left: 16x24x16 Middle: 24x36x24 Right: 32x48x32

impact as it is mainly apparent from the motion of the fire, but altering the resolution of the CFD grids has a significant effect on the animation of the fire. As is mentioned in [KESR06], increasing the resolution of the grids will decrease the effects of vorticity confinement and similar parameters. The result is that as grid resolution increases the fire animation will gradually become more uniform, until it eventually just resembles a simple cone. This is a significant problem, as the fire should not alter its behavior significantly when the detail level changes. While the behavior changes gradually, which makes it harder to notice, the difference in behavior between different grid sizes is large enough that a viewer is likely to notice it. However, the behavior also depends on parameters set for the flame. A way to counteract the changing behavior is to scale the parameters with the grid resolution. Doing so should preserve the overall behavior of the fire as the grid size changes.

The visual impact of lowering the particle count as the fire moves away from the camera is illustrated in Figure 6.5. As the particle count is lowered the particle size and intensity is increased accordingly. The particle counts for

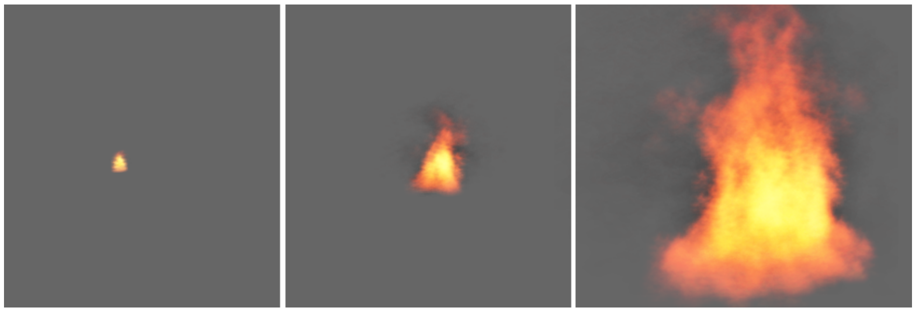

Figure 6.5: Screen shots of fire seen at different distances using different particle counts and grid resolution 16x24x16. Left: 20 particles. Middle: 200 particles. Right: 4000 particles.

the images in Figure 6.5 is lowered by a large amount as the fire moves away. The difference between the closest and farthest images is 3980 particles, or 99.5% of the total number of particles. However, even with so few particles the fire still displays the characteristic flickering behavior and overall appearance.

6.2.4 Texture disabling

Altering the particle textures as the detail level is lowered is included to further increase performance when the detail level is low. The performance impact of disabling particle textures with different particle counts is shown in Table 6.4. The performance impact of disabling particle textures is somewhat surprising, giving a bigger performance boost than anticipated even with relatively few particles. The performance impact is, as expected, lower when fewer particles are used, but even with only 256 particles performance is more than tripled.

The difference in visual quality when viewing the fire at different distances with particle textures disabled is shown in Figure 6.6. The visual result is also promising. Even at close range the quality of the fire is not much worse than when using textures, and further away it becomes very difficult to spot

Particle count	Textured	Not textured
256	317.39	1002.23
512	169.53	806.94
1024	93.01	575.79
2048	47.20	375.12
4096	24.63	215.85

Table 6.4: Performance impact of disabling particle textures. Grid resolution used is 16x24x16.

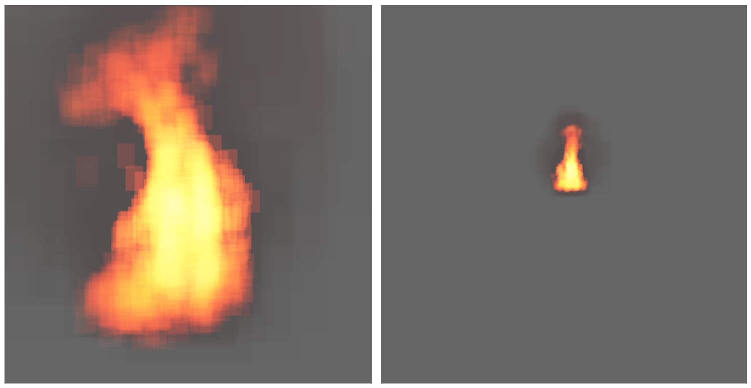

Figure 6.6: Fire particles rendered without textures using grid resolution 16x24x16 and 2048 particles.

the difference. The smoke, however, becomes noticeably blocky, even at a distance. The reason is that the smoke particles are rendered at a much larger size than the fire particles. When the particles are large, the square shape of the untextured particles becomes noticeable even at a distance.

The problem, then, is that while disabling particle textures gives a significant performance boost, the visual quality becomes a problem if the particle size is increased significantly. Disabling particle textures thus becomes a problem when combined with the change in particle count and size as the fire moves further away. However, since the performance boost from disabling particles is significant, it may be a good option to reduce the amount of change made to particle count and size. Disabling particle textures should make up for the performance loss from having more particles.

6.2.5 View culling

View culling is performed to increase the efficiency of the rendering when the fire is outside the current view. Performance impact from culling different components when the fire is in view can be seen in Table 6.5. It should be

Components running	16x24x16	24x36x24	32x48x32
Both	29.45	14.20	5.90
Visualization	37.12	37.13	37.21
Simulation	1301.32	265.10	7.54

Table 6.5: Performance when running one or both of visualization and simulation using different grid resolutions. Particle count is 2048 fire particles and 512 smoke particles.

noted that the simulation of the particles (updating of speed and position) are done in the simulation step of the rendering. The visualization step only renders the particles to the view. An interesting aspect to consider is the overhead of running the visualization. The leap from 37 to 1300 frames per second when disabling the visualization seems significant. A counter argument to this is that when running the visualization using a system of only a single particle, enabling or disabling the particle system seems to have no discernible effect. Compared to the results seen in Figure 6.5 this is a very promising result, since lowering the particle count gives a good performance boost, and the fire can still look convincing with few particles when it is far away. As noted earlier, however, the performance impact of the visualization is directly related to CFD grid resolution, as the exponential rise in simulation cost will quickly dominate the visualization cost.

Performance results for culling of different components when the fire is outside the view are seen in Table 6.6. As can be seen, merely looking away from the fire increases performance significantly (the fire runs at roughly 20 frames per second when visible and fully rendered). The overhead of passing the particles to the rendering pipeline is also visible in the difference between enabled and disabled visualization. While the internal culling in the graphics pipeline will reduce the cost, passing the particles to the point where they are culled still incurs a significant cost.

	Visualization on	Visualization off
Simulation on	445.83	1297.39
Simulation off	615.34	1729.11

Table 6.6: Frame rate impact of view culling when fire is outside view. Particle count is 4096 fire particles and 512 smoke particles, and grid size is 16x24x16.

6.2.6 Simulation step skipping

Simulation step skipping is performed to increase the performance of the simulation when the detail level of the fire is reduced. Skipping simulation steps means that the fluid simulation calculations are not performed every frame. If the fluid simulation is halted, the fire fluid would act as a static velocity field, which the fire particles could still flow through normally. The fire would appear as a simple static animation, however. The idea is to not perform the fluid simulation every frame, thus increasing performance while preserving the overall behavior for a fire rendered at low detail. The impact on performance from simulation step skipping can be seen in Table 6.7. With only the simulation running, the performance increases roughly

Frames skipped	Simulation only	Simulation and visualization
0	256.90	14.22
1	504.82	25.66
2	790.18	33.17
3	1063.92	39.63
7	1285.03	40.70
15	1313.31	41.23

Table 6.7: Performance impact of using simulation step skipping. Fire run with grid resolution 24x36x24, 2048 fire particles, and 512 smoke particles. Frames skipped refers to the number of frames rendered between each simulation step. Skipping three frames means running the simulation for every fourth frame, and so on.

linearly when between 0 and 4 frames are skipped. However, after that the performance increase drops sharply, with an apparent cap at around

1300 frames per second. It is likely that application overhead for running CFD shader program or other peripheral operations prevent the frame rate from rising much above this level. This assumption is supported by results found when using grid size 16x24x16, where the frame rate would be around 1300 with no frames skipped but never rise much, and also when using grid size 32x48x32, where the frame rate would continue to rise linearly longer, until reaching approximately the same value. The same trend is seen when the visualization is included, though the performance is lower due to the significant performance hit from running the visualization.

These results indicate that step skipping may only give good performance results when the CFD resolution is high. Unfortunately, skipping simulation steps slows down the progression of the fluid, making the animation look somewhat odd at close ranges (which is where using high grid resolutions is appropriate). Increasing the size of each time step can alleviate this problem, but with the existing code increasing the time step value by a factor greater than 4 tends to introduce instabilities and erratic behavior to the fire. The combination of these results leads to the conclusion that with the current code, time step skipping does not work particularly well. However, if the instability and performance cap problems are solved, this may be a good way to increase the performance of the fire.

6.3 Summary

The results from the different tests run on the partially completed dynamic LOD framework have been presented in this chapter. An outline of the most important observations made is given below.

- The LOD algorithm itself has little impact on performace, with only the texture resizing introducing a small overhead.
- The performance impact of the visualization's detail level depends on the simulation's detail level. At small CFD grid sizes, the performance scales roughly linearly with particle count. But as the CFD grids increase in size, the cost of the simulation increases exponentially, quickly dominating the cost of the fire rendering.
- Changing CFD grid size changes the overall behavior of the fire. This needs to be corrected by scaling fire parameters together with the CFD grid size.

- Reducing particle count while increasing particle size and intensity gives good visual results, with the characteristic flickering of the fire preserved at a distance even when the particle count is reduced from 4096 to 20.

- Disabling particle textures gives a larger performance boost than expected, and good visual results. However, the visual quality declines when particle size increases, giving a conflict with the technique of increasing particle size as the detail level is lowered.

- View culling gives very good performance increases by disabling fire visualization and simulation when the fire object is not in the current view.

- Simulation step skipping gives a good performance boost, but works best at high CFD grid sizes, and noticeably slows down the development of the fire. Attempting to increase step size to fix slowdown introduces instability to the fire.

While the results presented here are tentative rather than being conclusive proof that the algorithm works as wanted, the results are a strong indication that a fully implemented dynamic LOD algorithm will give good results both in terms of performance and visual quality.

Chapter 7

Discussion

In this chapter I discuss the results achieved during the work with this thesis. I discuss the problems encountered while trying to construct and implementation of the LOD framework; problems that led to my inability to fully finish the framework. The requirements presented in Chapter 1 are evaluated based on the capabilities of the framework presented in 4.6. A conclusion is then made based on the evaluation and experiences made while working. The primary contributions from my work to the field of graphics and LOD are presented. Finally, I present various ways in which the algorithm could be improved in the future.

7.1 General

I have presented a method for adding a dynamic LOD algorithm to a physically based fire rendering running on the GPU. The algorithm runs on the GPU where this is reasonable, and requires little resources to alter the detail level of the fire rendering. However, while the test results presented indicate that the algorithm will give good results in terms of performance and visual quality, I was unable to complete the implementation and get conclusive results.

As I was set to begin implementation, I made an unfortunate choice in that I decided to use the code from [KESR06], and attempt to change it to incorporate my LOD method, rather than start over. This was done because I initially believed this to be less work than to build a new fire rendering from scratch. The code I attempted to use was highly complex, difficult to read, and incompatible with what I wished to accomplish. Thus, much of the time was spent on what can best be described as "fiddling"; attempting to make changes to the code to incorporate the LOD algorithm and scenegraph functionality. This approach was eventually abandoned, but there was not sufficient time remaining to complete the fire rendering with LOD before the deadline. Part of the reason why I attempted to work on earlier code was that the size of the code and the fact that two people had worked on it made it seem likely that there may not be enough time to build the application from scratch. Closer examination later on, however, revealed that a considerable amount of the code was for peripheral functionality that was not necessary or even made the implementation of LOD harder. These problems can be attributed to a lack of effort on the prestudy for the thesis. Had more time been spent on studying the code and considering the different options rather than starting to code with little forethought, these problems might have been avoided and a proper implementation of the fire LOD algorithm completed.

In the end, the problems I encountered forced me to settle for running small tests for various parts of the framework, and evaluating the effect the different LOD operations would have on the fire rendering in terms of performance impact and visual quality.

7.2 Evaluation

The evaluation of the constructed framework is somewhat troublesome, as there are in fact two separate frameworks, both incomplete. Some functionality was completed for the framework that used Rødal and Storli's code as a basis, but it turned out that not enough time was available to make the necessary changes to their code to make it compatible with important functionality. The new framework contains some of the missing functionality, but there was not enough time to complete the fire rendering for proper testing. The evaluation of each requirement presented in Chapter 1.2 will consider whether the requirement is satisfied by the framework presented in Chapter 4.6.

- ***R1:* The framework should allow a 3D physically based fire rendering to run in real time:**
  The framework simply wraps a fire rendering based on Rødal and Storli's fire rendering in a single object. The fire still renders in real time, thus satisfying the requirement.

- ***R2:* The framework should allow the fire rendering to be contained in a scene that also contains numerous other complex objects:**
  The framework satisfies this requirement by containing the fire in an object which is placed in a scenegraph. The scenegraph can contain an arbitrary number of other objects.

- ***R3:* The framework should allow several fire renderings to exist in the same scene, each with individual detail levels:**
  As the fire object is treated as any other object in the scenegraph, any number of fire objects may be included in the scene. Each fire object has its own simulation, visualization, and detail level. The requirement is satisfied.

- ***R4:* The framework should allow the fire simulation to be run at varying detail levels to increase either performance or quality as needed:**
  The framework enables changing the detail level of the fire simulation by altering the size of the grids used for CFD calculations and by introducing simulation step skipping. View culling is used to disable the simulation when the fire is out of view. These techniques give good results in terms of performance and visual quality, though changes in fire behavior need to be counteracted. This requirement is satisfied.

- ***R5:* The framework should allow the fire visualization to run at varying detail levels to increase either performance or quality as needed:**
  The framework enables changing the detail level of the fire visualization by changing particle count and size, disable particle textures, and use view culling to disable the particle system and replace the dynamic lighting system with a simple proxy simulation when the fire is out of view. The different changes in detail level give good results in terms of visual quality and performance. This requirement is satisfied.

- ***R6:* The framework should dynamically alter the detail level of the fire to conserve resources while giving the greatest possible quality when the fire dominates the view:**
  Nearest neighbor filtering is used to dynamically alter the CFD grid size. The particle system detail level is dynamically altered by setting

a respawn ratio to alter the particle count. A simple stochastic system is used for the dynamic lighting proxy simulation. This requirement is satisfied.

- ***R7:* The framework should, as far as is possible, run on the GPU:**
  Running the entire LOD algorithm on the GPU is possible, but currently impractical. The reason is that for GPUs without geometry shaders every fire particle needs to be passed from the CPU to the GPU. The overall LOD calculation and setting of different parameters is therefore done on the CPU, while the most costly component, the CFD grid resizing, is done on the GPU. If the fire rendering is moved completely to the GPU, the LOD algorithm could also run completely on the GPU. This requirement is satisfied.

Because the implementation of the framework could not be completed in time, the evaluation presented here is tentative, based on the different test results presented in Chapter 6. In particular, the results of dynamically altering the detail level could not be tested.

7.3 Conclusion

Because the implementation could not be completed in time for the deadline the evaluation of the results are only tentative, and more work is required to determine if the LOD algorithm will truly give the desired effect on the fire rendering, allowing it to run well as part of a larger scene with a variety of other complex objects. As it is, the results of the tests performed are not definite proof that the algorithm works. They do, however, give a strong indication that the LOD algorithm will give good results, both in terms of performance and visual quality. The test results indicate that a complete implementation of the framework will satisfy the requirements specified in Chapter 1.2, though more thorough testing of a finished implementation is needed to determine how to apply the different LOD components to get the best result.

Some important observations have been made that should contribute to future work in this area. I have shown that there are significant performance gains to be made by adjusting the detail level of a fire rendering. I have also shown that when a fire rendering is of little importance to the current view, it is not necessary to render it at full detail, and that changes to the detail level can be performed on the GPU to avoid slowdown.

The largest unknown that remains is the dynamic change in detail level. A difficult aspect of working with LOD is how to make the transition from one detail level to another smooth. If the viewer moves a small amount away from a fire object, it should not be noticeable that the detail level of the fire has changed. Even if the difference between the smallest and greatest detail levels are noticeable, the difference for each small step should not be. Because the implementation was not completed, the visual impact of moving from one detail level to another could not be tested.

7.4 Contributions

We have presented a method for increasing the efficiency of a physically based fire rendering running on a GPU. The contributions of this work to the area of real-time rendering of physically based fire are as follows:

- A method for applying dynamic LOD to a physically based rendering of fire, tested with the fire rendering presented in [SR06].
- A method for including multiple fire objects, each with individual visualizations, simulations and detail levels.
- A method for dynamically altering the detail level of the fire's particle system, using particle respawning as a way of altering particle count, altering the size and intensity of each particle, and using texture disabling and view culling, in order to increase the performance of the fire rendering.
- A method for dynamically altering the detail level of a fluid simulation based on the Navier-Stokes equations, using nearest neighbor filtering to gradually scale the size of the CFD grids as well as simulation step skipping and view culling, in order to increase the efficiency of the fire rendering.
- The methods used for the different components of overall LOD calculation, visualization LOD, and simulation LOD are all independent, allowing for replacing each component to better suit the needs of an individual application.

The simulation LOD in particular uses a general technique that should be possible to use for most any fluid simulation that uses the Navier-Stokes equations on a grid of discrete cells. A work-in-progress article [OEG07] was submitted to and approved for presentation at Theory and Practice of Computer Graphics 2007. The paper was awarded the Terry Hewitt prize[1].

7.5 Future work

There are numerous possibilities for improving and expanding the LOD algorithm, and some of these are discussed below.

- While the test results presented here are an indication of the viability of having dynamic LOD for physically based fire renderings, a complete implementation of the LOD algorithm is needed for conclusive results.

- There is a significant variety of ways to perform the various LOD operations. While the most suited methods may vary from application to application, thorough testing of the different LOD conditions and formulas for calculating various LOD values should be performed to determine the advantages and disadvantages of each.

- The functionality offered by new GPU chipsets such as the Geforce 8 series should make it possible to move even more of the load of the fire rendering from the CPU to the GPU or find other ways to increase the efficiency of the rendering. Similarly, the new functionality may offer new and improved ways to perform the LOD calculations. The new geometry shader in particular should present an opportunity to move more of the rendering and attached LOD algorithm to the GPU, further reducing the amount of necessary communication between the CPU and GPU.

- The system of textured particles used by Rødal and Storli is one of several different ways to perform the visualization of the fire. Other visualization methods may be more costly to perform but offer more

[1]The Terry Hewitt Prize is awarded to the best technical research student paper on the basis of both the written paper and its presentation. All papers submitted by Master students and Ph.D students are eligible for this prize (http://www.eguk.org.uk/TPCG06/cfp.html).

realistic results. Examples include volume renderings or visualizations that project values to relatively simple polygonal surfaces. Such methods may also benefit considerably from dynamic LOD, allowing them to be implemented for real-time applications.

- As for visualizations, there are different possible ways to perform the simulation of the fire, such as doing the simulation without the grid of discrete cells. Finding ways to perform dynamic LOD for these simulation methods can expand the versatility and application possibilities for the algorithm.

- As mentioned previously, CFD calculations are used for many different types of application areas involving fluid dynamics. While the details differ, the basic principles of the different types of fluid calculations are very similar. The LOD algorithm could conceivably be expanded to many of these areas such as water simulations and more general gas simulations.

Bibliography

[AL02] Nick Foster Arnauld Lamorlette. Structural modeling of flames for a production environment. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 729–735, 2002.

[BDL03] Niels Jørgen Christensen Bent Dalgaard Larsen. Real-time terrain rendering using smooth hardware optimized level of detail. 2003.

[Blo88] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design, Volume 5*, pages 341–355, 1988.

[Che01] Stephen Chenney. Simulation level-of-detail. 2001.

[CO02] H. Vilhjálmsson J. Dingliana S. Dobby B. McNamee C. Peters T. Giang C. O'Sullivan, J. Cassell. Levels of detail for crowds and groups. *Computer Graphics Forum, Volume 21 number 4*, pages 733–741, 2002.

[DF97] Stephen Chenney David Forsyth. View-dependent culling of dynamic systems in virtual environments. *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 55–58, 1997.

[DO01] Ming C. Lin David O'Brien, Susan Fisher. Automatic simplification of particle system dynamics. *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*, pages 210–257, 2001.

[DQN02] Henrik Wann Jensen Duc Quang Nguyen, Ronald Fedkiw. Physically based modeling and animation of fire. *Proceedings of the*

29th annual conference on Computer graphics and interactive techniques, pages 721–728, 2002.

[DT03] Machiko Tamura Tadahiro Fujimoto Kazunobu Muraoka Norishige Chiba Daiki Takeshita, Shin Ota. Particle-based visual simulation of explosive flames. *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pages 482–486, 2003.

[Eri00] Carl M. Erikson. *Hierarchical levels of detail to accelerate the rendering of large static and dynamic polygonal environments.* PhD thesis, University of North Carolina, Chapel Hill, NC, 2000.

[FD05] George Drettakis Francis Schmitt Florent Duguet, Carlos Hernandez. Level of detail continuum for huge geometric data. *SIGGRAPH 2005*, 2005.

[FR06] Oscar Ripolles Carlos Granell Francisco Ramos, Miguel Chover. Continuous level of detail on graphics hardware. *Proceedings of the 13th International Conference on Discrete Geometry for Computer Imagery*, 2006.

[Hop98] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *Proceedings of the conference on Visualization '98*, pages 35–42, 1998.

[Hue03] David Luebke Martin Reddy Jonathan D. Cohen Amitabh Varshney Benjamin Watson Robert Huebner. *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, 2003.

[HV95] W Malalasekera H.K. Versteeg. *An introduction to Computational Fluid Dynamics, The Finite Volume Method.* Prentice Hall, 1995.

[JH97] Deborah Carlson Jessica Hodgins. Simulation levels of detail for real-time animation. *Graphics Interface '97*, pages 1–8, 1997.

[JJ05] Sheng Li Xuehui Liu Junfeng Ji, Enhua Wu. Dynamic lod on gpu. *Computer Graphics International 2005*, pages 108–114, 2005.

[JK05] Rüdiger Westermann Jens Krüger. Gpu simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum, 24(3)*, 2005.

[KESR06] Geir Storli Knut Erik Samuel Rødal. *Physically Based Simulation and Visualization of Fire in Real-Time using the GPU.* PhD thesis, Norwegian University of Science and Technology, Trondheim Norway, 2006.

[KP89] E.M Hoffert K. Perlin. Hypertexture. *International Conference on Computer Graphics and Interactive Techniques*, pages 253–262, 1989.

[Lev02] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. *Proceedings of the conference on Visualization '02*, pages 259–266, 2002.

[LG96] James K. Hahn Larry Gritz. Bmrt: A global illumination implementation of the renderman standard. *Journal of Graphics Tools, Vol. 1, No. 3*, pages 29–47, 1996.

[MB06] Hassan Foroosh Murat Balci. Real-time 3d fire simulation using a spring-mass model. *Multi-Media Modelling Conference Proceedings, 2006 12th International*, page 8pp, 2006.

[OEG07] Lars Tangvald Odd Erik Gundersen. Level of detail for physically based fire. *Proceedings of Theory and Practice of Computer Graphics*, 2007.

[PB01] Pierre Poulin Philippe Beaudoin, Sèbastian Paquet. Realistic and controllable fire simulation. *No description on Graphics interface 2001*, pages 159–166, 2001.

[Pha05] Matt Pharr(editor). *GPU Gems 2.* Addison-Wesley, 2005.

[PL96] William Ribarsky Larry F. Hodges Nick Faust Gregory A. Turner Peter Lindstrom, David Koller. Real-time, continuous level of detail rendering of height fields. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, 1996.

[RB98] Michael Destriau Roland Borghi. *Combustion and flames : chemical and physical principles.* Èditions Technip, 1998.

[RCG02] Richard E. Woods Rafel C. Gonzalez. *Digital Image Processing, Second Edition.* Prentice Hall, 2002.

[SAKR00] R. A. Crawfis S. A. King and W. Reid. Fast volume rendering and animation of amorphous phenomena. 2000.

[SB05] O. Franzke J. Kopf O. Deussen S. Behrendt, C. Colditz. Realistic real-time rendering of landscapes using billboard clouds. *Proceedings of Eurographics 2005, Volume 24*, 2005.

[SC98] David Forsyth Stephen Chenney, Jeffrey Ichnowski. Efficient dynamics modeling for vrml and java. *Proceedings of the third symposium on Virtual reality modeling language*, pages 15–24, 1998.

[SC01] David Forsyth Stephen Chenney, Okan Arikan. Proxy simulations for efficient dynamics. *Proceedings of Eurographics 2001*, 2001.

[Shi90] Peter Shirley. A ray tracing method for illumination calculation in diffuse-specular scenes. *Proceedings of Graphics Interface '90*, pages 205–217, 1990.

[SPC05] Peter Shirley Clàudio T. Silva Steven P. Callahan, João L. D. Comba. Interactive rendering of large unstructured grids using dynamic level-of-detail. *Visualization, 2005. VIS 05. IEEE*, pages 199–206, October 2005.

[SR06] Odd Erik Gundersen Samuel Rødal, Geir Storli. Physically based simulation and visualization of fire in real-time using the gpu. *Proceedings of Theory and Practice of Computer Graphics*, 2006.

[SRWH98] Hans-Peter Seidel Stefan Röttger Wolfgang Heidrich, Philipp Slusallek. Real-time generation of continuous levels of detail for height fields. *Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization*, pages 315–322, 1998.

[Sta99] Jos Stam. Stable fluids. *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, 1999.

[TG00] Christopher Peters Carol O. Sullivan Thanh Giang, Robert Mooney. Aloha : Adaptive level of detail for human animation. *Eurographics 2000 short paper proceedings*, pages 71–77, 2000.

[TKH04] Daut Daman Tan Kim Heok. A review on level of detail. *Computer Graphics, Imaging and Visualization, 2004*, pages 70–75, 2004.

[YZ03] Zhe Fan Arie Kaufman Hong Qin Ye Zhao, Xiaoming Wei. Voxels on fire. *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 36, 2003.

Appendix A

Result tables

The following result tables are included for completeness, though they were considered too superfluous to include in the results chapter.

Grid size	LOD running	No LOD
16x24x16	27.81	29.28
24x36x24	13.60	14.22
32x48x32	5.78	5.90

Table A.1: Performance impact of running combined portions of the LOD algorithm.

Grid size	LOD calculated	LOD not calculated
16x24x16	29.08	29.22
24x36x24	14.24	14.17
32x48x32	5.96	5.97

Table A.2: Performance impact of running the overall LOD calculations.

Grid size	View culling	No view culling
16x24x16	29.08	29.22
24x36x24	14.24	14.17
32x48x32	5.96	5.97

Table A.3: Performance impact of running the visibility evaluation.

Grid size	Step skipping	No step skipping
16x24x16	29.08	29.22
24x36x24	14.24	14.17
32x48x32	5.96	5.97

Table A.4: Performance impact of calculating the step skipping.

Grid size	Particle adjust	No Particle adjust
16x24x16	29.08	29.22
24x36x24	14.24	14.17
32x48x32	5.96	5.97

Table A.5: Performance impact of altering particle count.

Appendix B

Theory and Practice of Computer Graphics 2007 paper

The following paper was submitted to and presented at the TPCG07 conference.

EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Level of Detail for Physically Based Fire

Odd Erik Gundersen and Lars Tangvald

Norwegian University of Science and Technology, Trondheim, Norway

Abstract

In this paper, we propose a framework for implementing level of detail for a physically based fire rendering running on the GPU. The physics of the fire is simulated using a fluid solver and combustion modelling, and the fire is visualised using a particle system. Our preliminary results indicate that by adjusting the simulation domain and particle system, performance can be increased without noticeably degrading the fire visually when it is far from the camera.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism, Animation

1. Introduction

Computer games push the limits of real-time graphics, and current titles like *Gears of War*™and *The Elder Scrolls: Oblivion*™have stunning graphics. Still, there is a long way to go until virtual environments are mistaken for real ones. Generally, games that take place in cities look more realistic than games that take place in natural environments. Contrary to human-build structures, natural phenomena are very complex, and thus hard to visualise in a realistic manner, especially in real-time. Until natural phenomena are rendered in a convincing way, virtual environments will look exactly like that; virtual. Our belief is that rendering of natural phenomena should be based on the laws of physics. However, there are problems related to this philosophy as solving equations describing the laws of physics often are computationally very demanding. This is because either lots of small equations need to be computed or that the solutions only can be approximated numerically using methods needing several iterations.

Our focus is on realistic rendering of fire. As many other natural phenomena, like smoke, water, and explosions, fire can be simulated using computational fluid dynamics (CFD). CFD methods used for creating realistic flames have generally been too computationally demanding for real-time implementations [TOT*03] [LF02]. But as hardware is becoming increasingly more powerful, this is changing. In recent years, numerous methods for rendering realistic fire that run in real-time have been published [ZWF*03] [AH05] [BF06].

Although the papers referenced above describe real-time techniques for rendering physically based fire, they are not easily utilised in a virtual environment. This is because they use large amounts of the resources available to render the fire only, even when the fires are far away from the camera and are not important to the scene. Similar problems related to object geometry are solved using level of detail (LOD) algorithms. LOD algorithms seek to reduce the detail of object geometry without creating a visual difference. Reducing the detail of the objects leads to shorter processing time, which again leads to higher frame rates. However, there is a lack of research on LOD algorithms for fluid dynamics system in the literature. We aim to develop a framework that enables the inclusion of a physically based fire into a virtual environment.

Our goal is to enhance our previous work on real-time fires presented in [RSG06] and [GRS06] with a LOD algorithm that reduces the computational cost without creating a notable difference in visual quality. As both the simulation and visualisation is completely executed on the GPU, this is a requirement for our LOD algorithm too. The framework is still under development, and the results presented in this paper are preliminary. In spite of that, the results indicate that we are on the right track.

Our contribution. The collection of methods presented in this paper is a first step to towards a fully GPU implemented LOD framework for rendering physically based real-time fires. Solutions for both the simulation part and visualisation part of the fire rendering process are presented. Dynamically resizing the resolution of the simulation domain together with simulation step skipping are the methods proposed for reducing the computational cost of the simulation, while particle size adjustment and particle count resizing are used for reducing the cost of the visualisation step. The framework can easily be extended to work for animating both smoke and explosions.

This paper is organised as follows. After a brief overview of related work, an introduction of the fire rendering algorithm is presented. Then, the LOD framework is proposed followed by preliminary results. The paper concludes with summary and future work.

2. Related Work

As physically based fires traditionally have been too computationally demanding for real-time applications, several non-physical methods have been developed. The most prominent non-physically based method is presented in [Ngu04]. They use video-textured sprites for creating believable raging fires with smoke in real-time. In order to add variety to the flames, two flame animations are combined in various ways. Another non-physical approach is presented in [KCR00]. They use volume rendering in combination with a set of textures to visualize animated amorphous materials such as fire, smoke, and dust. Dynamics and illusion of motion are created through cycling the textures in each voxel. In [FMF06], a method that uses a photometric solid defining luminous intensities for a set of zentihal and azimuthal directions is presented. The intensities are stored in a 2D texture and by rotating this texture the fire is animated.

We have not been able to find any previous work on LOD algorithms used with physically based fires. There are however a plethora of literature published on LOD. LOD for graphics is generally divided into three different types of methods according to [LRC*03]. These are discrete, which uses different versions of objects generated before starting the application, continuous, which generates new versions of the object during run-time, and view-dependant, which generates multiple detail levels of the same object during run-time.

[HD04] presents a list of of conditions that determines the importance of an object to the current frame. Among these conditions are velocity of the objects, distance from camera, size, and whether the objects are completely or partially visible. LOD has been applied to object geometry [CCSS05] [DHDS05] including terrain [Hop98] [RHSS98], physics [HC97] [FC97], and autonomous behaviour of objects not controlled by the user, like computer played characters and effect from weapons [OCV*02]. [OFL01] presents a method for clustering particles together to increase calculation efficiency of particle systems.

3. Rendering Physically Based Fire

The fire rendering process is divided into two parts. First, the fire is simulated, and then the simulation is visualised. Simulation is the most computationally demanding process because it solves a fluid system. This fluid system evolves four fields controlling the temperature, the amount of exhaust gas, the amount of fuel, and the velocity in the simulation domain. The simulation domain is the limited volume where the fire can burn.

After simulation, the state of the fluid system is visualised using a particle system of textured particles. The particles flow through the simulation domain guided by the velocity field. For each voxel in the simulation domain, a fire colour is computed and stored in a table called the fire colour field. The particle's texture colour is looked up in the fire colour field based on the particle's position.

The rest of this chapter gives a brief overview of the fire rendering process. For a detailed description of the complete method, see [RSG06].

3.1. Simulating Fire

The fire is simulated by evolving a fuel gas field, an exhaust gas field, and a temperature field in co-evolution with a velocity field. These fields are governed by the Navier-Stokes equations and the combustion process, which converts fuel gas to exhaust gas and heat when the temperature exceeds a certain threshold. Buoyancy due to heat then causes the hot exhaust gas to rise, which in combination with vorticity confinement, cause the characteristic fire-like motion.

3.1.1. The Simulation Domain

We use a voxel data structure to represent the simulation domain and will refer to each unit as a cell. The simulation domain limits the volume where a fire is simulated. There are two different kinds of cells in the simulation domain; interior cells and boundary cells. Each cell contains a corresponding field value. When discretizing the fields into cells, the field values are defined in the centre of the cells and assumed to be uniform inside each one. As for boundary conditions, the boundary cells are set to 0 in the density fields and to the wind vector for the velocity field.

3.1.2. Velocity field

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

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

$$\nabla\cdot\mathbf{u} = 0 \qquad (2)$$

The first term on the right-hand side of equation 1 is the self-advection of the velocity causing velocity to move along itself. The second term, $-\nabla p$, is the pressure gradient causing velocity to move from areas of high pressure to areas of low pressure. The pressure field is used as a correcting term ensuring that equation 2 holds. Equation 2 is the non-divergence condition, which states that the velocity field should be mass conserving. The last term on the right hand side of equation 1 is the external force acting on the velocity field. The external force actually consists of several separate forces as shown in equation 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.

3.1.3. Fire density fields

The three separate scalar fields specifying the amount of fuel gas, exhaust gas, and heat distributed throughout the simulation domain are collectively referred to as the fire density fields. These three scalar fields are evolved by the same equation:

$$\frac{\partial d}{\partial t} = -\mathbf{u} \cdot \nabla d + \kappa_d \nabla^2 d - \alpha_d d + S_d + C_d \tag{4}$$

The parameter d is a scalar quantity that represents either the amount of fuel gas, exhaust gas, or temperature in a cell in the simulation domain; denoted by g, a or T respectively. Equation 4 describes the evolution of a scalar field over time in the simulation domain as the velocity field $\mathbf{u}$ affects the scalar field. We use a slightly modified form of the equations described in [Sta99].

The first term on the right-hand side in equation 4 governs the advection of the scalar quantity d by the velocity field $\mathbf{u}$, while the second term governs the diffusion of the scalar quantity d. κ_d is the diffusion constant controlling the amount of diffusion associated with each of the density fields. Furthermore, the third term governs the dissipation of the scalar quantity d where α_d denotes the dissipation rate. The dissipation rate ensures that fuel gas, exhaust gas, and temperature will decrease over time. S_d denotes a source term used for increasing the scalar quantity d. Only the fuel gas field has a source, which is used for injecting fuel, while temperature and exhaust gas are produced solely in the combustion process. C_d is the combustion term that controls the effect of the combustion process on a specific density field.

3.2. Visualising Fire

Using a precomputed black-body radiation lookup table, a fire colour field is computed based on the exhaust gas and temperature fields. The fire is visualised using a particle system, and the particle positions are updated based on the velocity field, and the particle colours are read from the fire colour field. Smoke is implemented in a separate particle system, and the light intensity is based on the fire colour field.

3.2.1. Computing the fire colour field

We use Planck's formula for black-body radiation (equation 5) in order to calculate the intensity radiated by the hot exhaust gas.

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

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 colour should have a limited dynamic range suitable for display on traditional computer monitors. To map the given intensities between 0 and 1, we use the exponential mapping function from [Mat97]:

$$n = 1 - e^{\frac{-L}{L_{average}}} \tag{6}$$

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 5 and 6 are used to precompute black-body radiation colour 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 colour field. This is done for each cell in the simulation domain. Equation 7 shows how the colour $\mathbf{c}$ in the fire colour 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.

$$\mathbf{c} = a \times lookup\left(T_{scale} T\right) \tag{7}$$

3.2.2. Visualisation using two particle systems

We visualize the fire and the smoke using separate particle systems defined in the simulation domain. By computing a separate smoke field instead of trying to incorporate the smoke into the fire colour field, we get more control over the appearance and the amount of smoke produced in the fire. Each particle represents a small element of the fire or the smoke and has a set of associated variables: spawn position, current position, initial spawn delay, current velocity, and colour. Spawn position and initial spawn time are given at the beginning of the simulation, whereas the other variables are dynamically updated. A particle's colour is specified by an RGBA colour value. The amount of smoke is computed based on the temperature at a given cell and the amount of exhaust gas, quite similar to how the fire colour field is generated.

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:

$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{v}_i \delta t, \tag{8}$$

where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle i respectively and δt is the timestep. Based on the particle position, the particle's velocity and colour are found by interpolating samples from the discretized simulation velocity and fire colour 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 to create more low-level detail.

3.3. Additional Properties

Fires interact with their surroundings. Our fire implementation reacts to dynamic wind, looks realistic when moved, and illuminates surrounding objects. Wind is generated dynamically by utilising Perlin noise curves [Per85]. The wind vector operates on the simulation domain. Simulation domain advection is used for moving the fire around. The simulation domain and the particles are advected using the distance vector, which is the distance between the new and old position.

In addition, lights with dynamically set intensities are implemented. One or more point light sources are placed inside the simulation domain, and based on their position in the simulation domain the light intensities are computed from the amount of exhaust gas and temperature at their respective position. The dynamic lighting produces the flickering light often associated with fires.

4. Level of Detail Framework

As shown above, rendering of fire is a complex task with several steps utilising different technologies. Thus, there are several possible ways to reduce the computational load. We propose four different strategies, and they are:

Simulation domain resizing: The resolution of the simulation domain is changed according to the distance between the fire and the viewer. At close range, the simulation is done at maximum resolution, while reduced with prolonged range.

Simulation step skipping: By skipping simulation steps when the fire is not significant to the view, lots of resources can be saved. The visualisation steps are *not* skipped though.

Particle count adjustment: The amount of particles in the particle system is reduced when the camera moves away from the fire and increased when moving towards it.

Particle resizing: The size of the particles is increased when reducing the particle count and decreased when the particle count rises.

The following sections describe in detail the different strategies used in our LOD algorithm for physically based fire rendering.

4.1. Overall LOD

The overall LOD of an object is a value describing how detailed the object should be rendered. Two LOD conditions determines the overall LOD, and these are view culling and distance.

View culling determines whether the fire is inside the view or not. The angle between the camera's view direction and the line from the camera to the fire is calculated. If the angle exceeds a certain value, the fire is determined to be outside the view, and no further calculations are performed.

The distance between the camera and the fire is calculated. The detail level of the fire decreases linearly as the distance increases. This strategy is based on the assumption that a fire seen at a distance need less details to be visually convincing. The LOD value for distance is calculated by the formula:

$$lod = \frac{b}{d}, \tag{9}$$

where b is the maximum distance the camera can be away from the fire while still rendering the fire at full detail, and d is the distance between the fire and the camera.

4.2. Simulation Domain Resizing

The most computational demanding step in our fire rendering algorithm is solving the fluid dynamics system. Therefore, reducing the amount of computations associated with solving the fluid dynamics system is most important. Our solution is to resize the simulation domain according to how important the fire is to the scene. By reducing the size of the simulation domain, the Navier-Stokes equations are solved fewer times for each simulation step, and thus the fire need less resources to render.

The resolution of the simulation domain is changed according to the distance between the fire and the viewer. At close range, the simulation is done at maximum resolution, while it is reduced when prolonging the range.

To transfer values from the old simulation domain to the new, a form of bilinear filtering for three dimensions is used. Each cell in the new simulation domain is given the average value of the eight closest cells from the old simulation domain.

4.3. Simulation Step Skipping

In [RSG06], the simulation is done at each time step. However, when the flame is not significant to the view, it is not necessary to perform these calculations for every frame. Thus, simulation steps may be skipped while letting the particles used for visualisation travel through the simulation domain fetching old values from the velocity and fire density fields. This may, however, lead to a static looking fire if too many steps are skipped as the simulation in practice is slowed down.

There are other possible variants to this strategy. One is to extend the interval between each simulation step while visualising at constant intervals. This will probably lead to a more flickering fire, but the simulation is not slowed down and therefore the fire will not look static. The computational savings will not be as good as for step skipping as long as the intervals between simulation runs are less than a full step. However, the visual quality of the fire animation might be better. It might, however, be hard to implement a fully working version of this variant for the GPU.

The other variant of this strategy is a more low-level one. In stead of skipping or extending the interval between simulation steps, it is possible to lower the quality of the simulation. The velocity field simulation may be skipped every other simulation step, and the velocity field computed last simulation step may be used to evolve the fire density fields in the current simulation step. This will, for each skipped velocity field simulation, save the computational cost of solving twenty iterations of the Jacobi method, which are used for solving the advection step in the velocity field. Still, the Jacobi method is used to solve the diffusion step in all three density fields. Informal tests we have done show that reducing the diffusion approximation from twenty to four iterations may give satisfying visual results. This strategy will not save as many GPU cycles as the other two variants, but the visual quality should be better.

4.4. Particle Count Adjustment

In [RSG06], the size of the particles and the particle count is constant. When a particle is no longer visible it respawns at the base of the fire. When the fire is small or viewed briefly, a lower particle count is needed as the details of the fire is less important for the visual quality of the fire. There are several possible ways of adjusting the particle count.

As in [OFL01], particles with similar position and speed may be clustered together when computing the motion of the particle and visualised individually. Clustering particles may be a good solution when the motion of the particles are computed individually. The particles in our particle systems fetch their velocity from the velocity field, and their new positions are found from their velocity and current position. This is done in parallel for several particles at once (how many depend on your specific GPU). Another reason for gains in the frame rate when rendering fire on the GPU is that the fire particles are not connected in any way. The particles need not to know anything about other particles and thus only need to read from its own location in memory. We have not implemented this strategy.

Another option is to cluster neighbouring particles into one particle that behaves as one both for simulation and visualisation. As with the other method mentioned above, this would require knowledge about other particles, which would decrease the gain of performing the calculations on the GPU. Therefore this method has not been investigated further either.

We use particle respawning together with the cameras distance from the fire to control the particle count of the particle system. We define a target respawning variable that is set based on the distance between the fire and the camera. The target value is set to all particles in the particle system when the camera is close to the fire. The farther away the camera is from the fire, the lower target value. Particles are respawned until the target value is reached. If the particle count of the particle system is higher than the target value, no particles are respawned.

Halting respawning would create a region of the flame with few or no particles followed by a wall of flame as respawning restarts. To counteract this, spawning can be done according to the ratio of current and desired particle count. For example, if the particle count is twice the target count only every other particle will respawn until the desired count is reached.

The target particle count decreases linearly with the fire's overall LOD value. The target count t is determined by the formula

$$t = basecount * lod, \tag{10}$$

where *basecount* is the base particle count used at the highest detail level and *lod* is the overall LOD value. Particles are added to or removed from the fire by altering the respawn rate of particles instead of adding or removing particles instantly.

4.5. Particle Resizing

When reducing the number of particles in the particle system, the fire may look less dense as the particles are quite small. Also, strange looking holes may appear in the flame. To counterbalance this, we adjust the particle size according to particle count. If the number of particles is lowered, the size of each particle is increased in scale to compensate. Altered particle size only affects newly spawned particles.

Particle size increases logarithmically as the fire's overall LOD value decreases. The size of particles s is determined by the formula:

$$s = \log d, \tag{11}$$

where d is the distance from the camera.

4.6. Other Considerations

For many physical simulations it is important to approximate the behaviour when not in view. For example, you expect a ball or a heat-seeking missile to have a certain behaviour when not in your view. If a ball is thrown out of your view and bounces on a wall, you expect it to return in the same direction. Also, if a heat-seeking missile is aimed and launched at you, you will not stop running just because you cannot see it any longer. Because of the turbulent behaviour of a fire, you will not have any visual expectations of how the flames have evolved when not looking at them. Thus, visual expectations of the viewer will not cause any problems.

There is a problem connected to illumination, though. The fire illuminates its surroundings and the light intensity is set based on the fire's properties. Something will have to be done to approximate the light intensity of the fire as it may be possible to look at the illuminated surroundings and not the fire. The fire intensity may be stochastically generated based on the maximum, minimum and mean temperature of the fire. In transitions between simulated light intensity and randomly generated, the light intensity will be interpolated between the last random generated and current simulated value for a short time interval. The light intensity will be generated by a proxy simulation when the fire is not in the view.

4.7. The Complete Algorithm

The LOD algorithm shown in Figure 1 is run for each frame. The algorithm first checks if the fire is visible. If not, the visualization is disabled, and a simple stochastic simulation is used to maintain the flickering of the dynamic lights. If the fire is visible, a single LOD value is calculated based on the fires distance from the camera as well as a factor determined by the scale of the scene. The LOD value is used to calculate a new size for the simulation domain and a new particle target count and spawn ratio. The simulation domain resizing is done by using a three dimensional form of bilinear filtering. When a fire particle is set to respawn the spawn ratio determines whether no, one or several particles are spawned.

```
calcLOD()
   if fire object does not
   intersect view frustum
      enableProxySimulation();
      disableParticleVisualization();
      return;

   oldLOD = newLOD;
   newLOD = min(distance(), 1);
   if newLOD == oldLOD
      return;
   newGridSize = maxGridSize * newLOD;
   resizeGrid(newGridSize);
   stepSkipping = 1 / newLOD;

   targetParticleCount =
      maxParticleCount*newLOD;

   particleSpawnRatio =
      targetParticleCount
      /currentParticleCount;

distance()
   dist = abs(cameraPosition
         - firePosition);
   return LODFactor / dist;

resizeGrid(newGridSize)
   for each newcell in new grid
      val = 0;
      find intersection in old grid closest
         to newcell's position

      for each oldcell
      in old grid bordering intersection
         val += oldcell;
      newcell = val / 8;

particleRespawn()
   if (currentParticleCount
   != targetParticleCount)
      counter = counter + particleSpawnRatio;
      while (counter > 1)
         spawn single particle;
         counter = counter - 1;
         currentParticleCount++;
      Adjust particle parameters;
   else
       respawn single particle;
   currentParticleCount--;
```

Figure 1: *Pseudo code for the LOD framework for our physically based fire implementation.*

5. Results and Evaluation

All tests were run on an intel 1.83GHz Core Duo with 1GB RAM and an NVIDIA Geforce 7600 Go with 512 MB VRAM. Three different tests have been implemented. The first one focused on possible LOD methods and ran without a proper fluid simulation. Simulation domain resizing, step skipping, particle resizing, and particle count adjustment was implemented. The performance gains indicated by these results did convince us that we were on the right track.

Table 1 shows the result of the second test, which was a 2D fire rendering running on the CPU using different grid dimensions. As can be seen, the performance increases linearly with the size of the grid, with performance roughly quadrupling when the dimensions are halved. The resulting flame is shown i figure 3.

5.1. Results

Grid size	Frame rate
128x128	12.76
64x64	51.25
32x32	202.54
16x16	798.68

Table 1: *Performance results for the second test: two-dimensional flame using different simulation domain sizes.*

Table 2 shows the performance result from the third test. A 3D fire that ran completely on the GPU with different grid dimensions and particle counts was tested, see figure 2 for the visual result. The tests were run with the camera a constant distant from the fire. As expected, the impact from altering the particle count is greater when the grid dimensions are small, as the simulation dominates the calculation complexity when the dimensions are larger. This indicates that altering particle count needs to be combined with altering the simulation dimensions to be effective.

	Grid size		
Particle count	16x24x16	24x36x24	32x48x32
512	43.27	20.87	6.28
1024	37.12	18.19	6.12
2048	29.20	14.19	5.94
4096	20.11	10.07	5.54
8192	12.37	7.86	4.87
16384	7.64	5.78	3.93

Table 2: *Performance results (frames per second) for the third test with a three-dimensional flame using different simulation domain sizes and particle counts.*

6. Summary and Future Work

We have presented an algorithm for combining dynamic LOD with physically based fire rendering on the GPU. The algorithm is based on changing the size of the simulation domain and altering the particle system to increase performance of the fire rendering when the fire is far away or not visible. While the work is still incomplete, the preliminary results presented indicate that the algorithm should give good performance gains without significantly degrading the visual appearance of the fire when it is far away from the camera.

Future work will include implementing the complete framework for execution on the GPU to get conclusive results.We will also investigate additional ways to calculate the relative importance of the fire, for instance determining whether other objects obscure the camera's view of the fire.

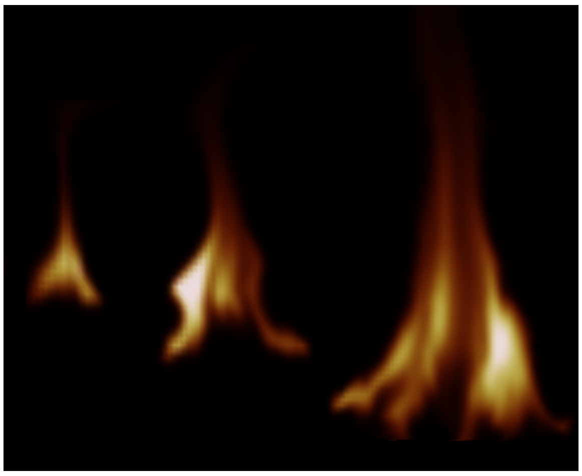

Figure 3: *Screen captures from the second test. The figure shows three different 2D fire renderings with simulation domain dimensons 32x32, 64x64 and 128x128.*

References

[AH05] Adabala N., Hughes C. E.: Grid-less controllable fire. *Game Programming Gems 5 (K. Pallister, Ed.), Charles River Media* (2005), 539–549.

[BF06] Balci M., Foroosh H.: Real-time 3d fire simulation using a spring-mass model. *Multi-Media Modelling Conference Proceedings, 2006 12th International* (2006), 8pp.

[CCSS05] Callahan S. P., Comba J. L. D., Shirley P., Silva C. T.: Interactive rendering of large unstructured grids using dynamic level-of-detail. *Visualization, 2005. VIS 05. IEEE* (October 2005), 199–206.

[DHDS05] Duguet F., Hernandez C., Drettakis G., Schmitt F.: Level of detail continuum for huge geometric data. *SIGGRAPH 2005* (2005).

[FC97] Forsyth D., Chenney S.: View-dependent culling of dynamic systems in virtual environments. *Proceedings 1997 Symposium on Interactive 3D Graphics* (1997), 55–58.

[FMF06] F. B.-L., M. L., F R.: Afigraph Õ06: Enhanced illumination of reconstructed dynamic environments using a real-time flame model. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, and interaction in Africa* (Aire-la-Ville, Switzerland, Switzerland, 2006), ACM Press.

[GRS06] Gundersen O. E., Rødal S., Storli G.: Physically based simulation and visualization of fire in real-time using the gpu. In *Eurographics UK Chapter*

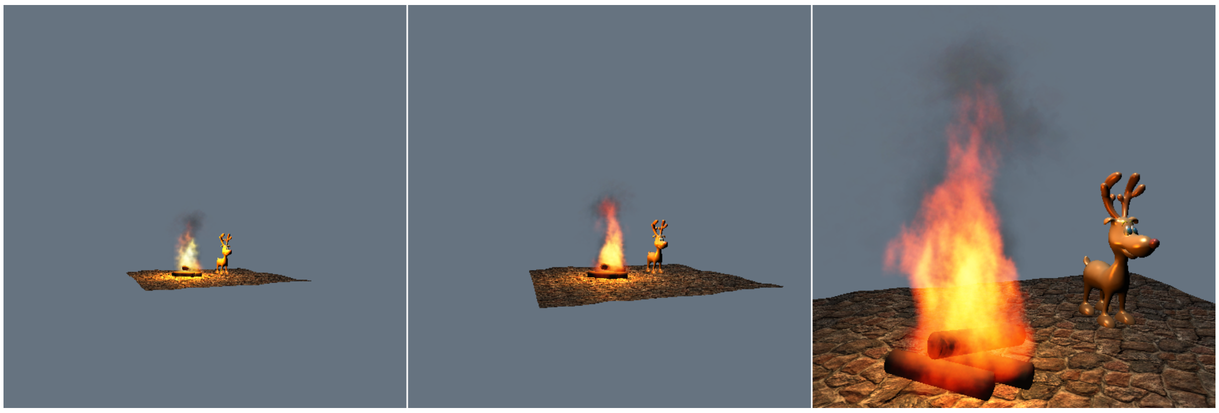

Figure 2: *Screen captures from the third test. Fire rendering at different distances using 20, 200, and 4000 particles and simulation domain dimensions 16x24x16.*

Proceedings: Theory and Practice of Computer Graphics 2006 (Aire-la-Ville, Switzerland, 2006), Eurographics Association, pp. 13–22.

[HC97] HODGINS J., CARLSON D.: Simulation levels of detail for real-time animation. *Graphics Interface '97* (1997), 1–8.

[HD04] HEOK T. K., DAMAN D.: A review on level of detail. *Computer Graphics, Imaging and Visualization, 2004* (2004), 70–75.

[Hop98] HOPPE H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. *Proceedings of the conference on Visualization '98* (1998), 35–42.

[KCR00] KING S. A., CRAWFIS R. A., REID W.: Fast volume rendering and animation of amorphous phenomena.

[LF02] LAMORLETTE A., FOSTER N.: Structural modeling of flames for a production environment. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), 729–735.

[LRC*03] LUEBKE D., REDDY M., COHEN J. D., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, 2003.

[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.

[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.

[OCV*02] O'SULLIVAN C., CASSELL J., VILHJÁLMSSON H., DINGLIANA J., DOBBY S., B. MCNAMEE C. PETERS T. G.: Levels of detail for crowds and groups. *Computer Graphics Forum, Volume 21 number 4* (2002), 733–741.

[OFL01] O'BRIEN D., FISHER S., LIN M. C.: Automatic simplification of particle system dynamics. *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings* (2001), 210–257.

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

[RHSS98] RÖTTGER S., HEIDRICH W., SLUSALLEK P., SEIDEL H.-P.: Real-time generation of continuous levels of detail for height fields. *Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization* (1998), 315–322.

[RSG06] RØDAL S., STORLI G., GUNDERSEN O. E.: Realistic 2d fire in real-time. In *Norsk Informatikkonferanse NIK 2006* (Trondheim, Norway, 2006), Tapir Forlag, pp. 189–200.

[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.

[TOT*03] TAKESHITA D., OTA S., TAMURA M., FUJIMOTO T., MURAOKA K., CHIBA N.: Particle-based visual simulation of explosive flames. *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on* (2003), 482–486.

[ZWF*03] ZHAO Y., WEI X., FAN Z., KAUFMAN A., QIN H.: Voxels on fire. *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), 36.