Modelling, Animation, and Visualisation of Fire

Theo Engell-Nielsen <beyond@beyond.dk>

Søren Trautner Madsen <trauma@diku.dk>

DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Denmark

Thesis submitted in conformity with the requirements
for the Degree of Master of Computer Science at the
University of Copenhagen, Denmark

6th April 1999

Abstract

The art of reproducing natural phenomena like water, smoke, and fire using only a computer is said to be among the most difficult. Of these phenomena, fire is perhaps one of the most difficult to understand. This makes it difficult to model and therefore very difficult to visualise. To obtain a complete reproduction of fire, we are brought to the boundaries of human cognition, computational complexity, and computer memory usage.

The path to photo-realistic depiction of fire historically spans quite a long trail. Particle systems were first defined by Reeves, Blinn contributed with a rendering method for clouds, scattering was introduced by Rushmeier and Torrance, progressively refined radiosity was defined by Wallace, Stam and Fiume used diffusion processes and stochastic modelling and rendering, and, finally, we end up with the recent works of Foster and Metaxas, concerning modelling the forces that affect the motion of hot, turbulent phenomena.

This thesis' contribution consists of an overview of the topic and a discussion of the related literature concerning modelling, animation and visualisation methods. A combination of the methods we find best is implemented into a simple raytracer in order to bring forth realistic looking, real-life constrained fire and smoke. While doing this, we have sought to reduce the amounts of parameters needed when defining fire in the raytracer.

Since literature on the subject is scarce we furthermore hope that this thesis will improve the availability of knowledge about digital fire, and that it will be a good introduction to the subject for those with an interest in modern computer graphics.

KEYWORDS: Fire, smoke, computer graphics, modelling, animation, rendering, visualisation, particle systems, blobs, turbulence, backwards warping, CFD, computational fluid dynamics, Navier-Stokes, voxels, radiosity.

For Peter & Jette

Contents

List of Figures

List of Tables

Notation

Notation	**Explanation**
A_i	Area of surface i.
β	Thermal expansion coefficient.
β_0	Relaxation coefficient.
B_i	Radiosity of surface i.
$\mathbf{c}$	Blob centre.
$\delta(\ldots)$	A density function.
E_i	Emitted energy per unit area of surface i.
$\Phi(s,t)$	A turbulence energy spectrum function.
$\mathbf{F}_{bv}$	Thermal buoyancy.
$\mathbf{F}_{ext}(\mathbf{x},t)$	Sum of external forces in position $\mathbf{x}$ at time t.
F_{ij}	Form-factor from surface i to surface j.
$\mathbf{g}_v$	Gravity vector. Only the vertical component is included.
$\hat{g}(\omega)$	$g(x)$ fourier transformed.
$\hat{\mathbf{H}}(\mathbf{s},t)$	A filter useful for transforming white noise to turbulent noise.
(i,j,k)	Position in the voxel grid.
k	Amount of blob sample steps.
λ	A scalar, a scaling factor.
l	Amount of light sources.
$N(\ldots)$	Array of white noise.
ω	A frequency variable.
ψ	Potential field (mass discrepancy).
p	Pressure.
ρ_i	Reflectivity of surface i.
$\rho(\mathbf{x},t)$	The basic blob density function.
r	Radius or distance.
$R(\ldots)$	A random field.
$\Delta\tau$	Voxel side length.
t	Time variable.
T_0	Initial reference temperature.
$T_{i,j,k}$	Voxel temperature.
$\mathbf{u}=(u,v,w)$	Motion vector in a motion field or voxel.
ν	Kinematic viscosity—the 'thickness' of a gas.
w	Amount of blob backwards warp steps.
$W(\ldots)$	Model-directed synthesis random function.
$\mathbf{x}=(x,y,z)^T$	A location in $\mathbb{R}^3$, the x, y, and z-coordinate.

Glossary

Word	**Explanation**
Advection	Motion in a gas or fluid caused by external forces (such as a fluid being stirred by a spoon).
Albedo	The overall reflection in one or many directions of an object.
Animated motion field	See *Motion field*. An animated motion field is an automatically calculated motion field that is recalculated for each *frame* in an animation sequence.
Array interpolation	This term covers linear interpolation between two values in an array A, when the index into the array is not in $\mathbb{N}$, but in $\mathbb{R}$. This definition is also applicable for *voxel environments*.
Backwards warping	The algorithm that handles distortion of *blobs* caused by a *turbulent wind field*.
Backwarping	See *Backwards warping*.
Bitmap splatting	A method of pasting small bitmaps onto the display.
Black body palette	A palette which fades from black through red through orange to white.
Blob	A visualisation tool. Basically a centre and a *density field*.
Cellular automata	A simplified model of bacteria living and interacting in a dish. Most famous is perhaps the computer game *The Game of Life*.
CFD	Computational Fluid Dynamics. Calculating gas- or fluid motion by approximating the scene by a *voxel environment*.
CGI	Computer Graphics/Generated Imagery.
Convection	Gas or fluid flows inside a medium caused by heat differences.
Density Field	A field with a density function $\delta(x, y, \ldots, z) \curvearrowright \mathbb{R}\, x, y, \ldots, z \in \mathbb{R}$.
Field	A discrete subdivision of a two dimensional or three dimensional scene. Each subpart is given certain properties.
FFT	The Fast Fourier Transform. Mathematical method to transform a function from the time domain to the frequency domain (the spectral representation).

Word	**Explanation**
Form factor	The fraction of energy leaving one surface that arrives at another surface.
Frame	A single image from an animation. Animations shows typically 25 images per second.
Gaseous phenomenon	Term for a three-dimensional non-solid object, which could be dust, a cloud, smoke, or fire.
Gaussian random number	an instance of a random variable with a Gaussian distribution. Such a number can be approximated by adding N random numbers from the interval $[0, 1]$. If $N/2$ is then subtracted the mean will be zero. The number can then be scaled to fit in any desired range.
Mass divergence	The difference in mass that enters and leaves a voxel. Also called mass discrepancy.
Motion blur	When objects *move* relatively fast when captured on film, they become *blurred*, hence *motion blur*.
Motion field	A field of motion vectors. Useful for animating *particles*.
Opacity map	A (bit)map that corresponds to a texture. It defines the *translucency* of the texture at each location.
Participating media	A volume of gas with scattering or absorbing properties. The air in a dusty room is an example.
Particle	A primitive object with a couple of attributes. Example attributes could be: Position, size, colour, velocity, and so forth.
Particle emitter	A graphics primitive with the main purpose of creating, managing, and killing *particles*.
Particle system	A graphics primitive, that handles *particle emitters* and *particles*.
Radiosity	The inter-reflectivity of light between objects.
Radiosity light map	A sampling of the *radiosity* on a surface. The sampling is performed on a grid.
Random function	A function of which nothing is demanded except that $f(\mathbf{x}) = f(\mathbf{y})$ when $\mathbf{x} = \mathbf{y}$. $f : \mathbb{R}^n \curvearrowright \mathbb{R}^m$.
Scattering	When entering energy (i.e. light) is spread within an object.
Second level statistics	The fine details of an object. What you should see when zooming in on the object.
Second level turbulence	Artificial turbulence used with a *gaseous phenomenon* to add detail. This way it is used as *second level statistics*.
Self shadowing	The ability of a *gaseous phenomenon* to occlude light within itself.

Word	**Explanation**
Thermal buoyancy	The effect of hot air rising.
Translucency	The fraction of light that passes through an object. An opaque objects has a translucency of 0, while a completely transparent object has a translucency of 1.
Turbulent wind field	A field of motion vectors. These vectors are chosen to give a turbulent motion to a *particle* being animated by the field.
Volume rendering	Visualisation of objects or a world represented in three dimensions.
Volume shading	See *volume rendering*.
Voxel	A *vo*lume pi*xel*.
Voxel environment	A rectangular grid of voxels describing the properties of a scene.

Foreword

This thesis is written by Theo Engell-Nielsen and Søren Trautner Madsen, DIKU[1], in conformity with the requirements for the degree of Master of Science in the University of Copenhagen. Supervisor is Ph.D. Knud Henriksen, DIKU.

This thesis was completed 6/4-1999.

Reader's prerequisites

The reader of this thesis should be a computer science student that has passed a introductory university course on computer graphics. Alternatively it is sufficient to be familiar with [Foley et al., 1990] and raytracing as it is treated in [Watkins et al., 1992]. This implicates that the reader should not be afraid of advanced math, since this thesis does include fairly large amounts of it. The reader should know how to differentiate and integrate multi-dimensional functions and it will be helpful to be familiar with the FFT (Fast Fourier Transform) algorithm.

We will furthermore assume the reader has a layman's knowledge of fire.

Reading guide

Just before this foreword we have included a list of Notation and a Glossary. The list of Notation serves as a quick reference to the mathematical symbols that we have used. The Glossary is a small list of terms that we use, some of which may be new terms for the reader. If a term is left undefined in the text, it should be found there. A term written in *italic* is either a term, which will be introduced in the text or a not directly related term which can be found in the referenced literature.

In the figure captions we have inserted the dagger symbol (†) to denote that the figure has been included in the appendix containing enlarged versions of the presented images (appendix B).

Througout the presented examples in this thesis we use a right hand coordinate system when describing spatial models.

[1] DIKU = *Datalogisk Institut, Københavns Universitet.* Institute of Computer Science (datalogy), University of Copenhagen.

Acknowledgements

Søren wishes to thank his girlfriend for letting him spend way too long fiddling with this thesis instead of getting out in the real world and start earning some money!

Theo wishes to thank his girlfriend for putting up with him and these last months filled with postponed obligations and tiring work. I furthermore wish to dedicate this work to my very good friend Peter Nyman Hansen and my mother, Jette. My thoughts still visit you very often and I'm sorry that both of you are not here to participate in the completion of goals set a long time ago.

Thanks to all the people being "jailed" at the instructor's room and in the penthouse computer room at `DIKU` for putting up with us, helping us improve our eloquence and LaTeX abilities. Especially Morten Nicolaj Pedersen, Sofus Stampe Mortensen, Jakob Grue Simonsen, Sebastian Skalberg and Bo Kjellerup need to be thanked.

Thanks to all the people at the image group at `DIKU` for helping us out from time to time, guiding us from office to office, to dig out information on math, physics and references and finally for lending us enough disk space for putting this entire thesis together.

Thanks to many of the authors in our bibliography for the electronic correspondence we have had, without which we would have spent quite some hours fiddling with all sorts of tiny, tiny details. These Include: John Carmack, Kevin Musgrave, Nick Foster, Roger Doonan, Kevin McGrattan and David S. Ebert.

For proof-reading this thesis lots of thanks are due to Uffe Friis Lichtenberg and especially Martin Koch who had the arduous job of turning our attempts at the English language into readable prose. Hopefully they have removed all the serious errors, but any that might be left are wholly our responsibility. Any such errors remaining should be considered as a challenge for the reader to find and kindly report to us.

Finally, thanks must go to our supervisor, Knud Henriksen, for guiding us through the world of computer graphics and helping us complete this thesis.

Theo Engell-Nielsen Søren Trautner Madsen

DIKU, 6th April 1999.

Chapter 1

Introduction

"We're tired of your phony fireworks!"

—Cpt. Kirk, Star Trek (1966)

Animation and depiction of fire is not an easy task. The problems arise mostly due to the fact that we do not have a complete comprehension of fire and its complexity. Therefore it is very difficult to describe it.

It is said that producing a convincing depiction of fire is among one of the most difficult and most attractive problems within computer graphics. Since [Reeves, 1983] and [Blinn, 1982a] computer rendered fire have been close to something we accept as being fire but nevertheless is quite far from the real thing. What does it actually take to make these photo-realistic pictures of fire? The purpose of this thesis is to come closer to the answer to this question.

1.1 Motivation & background

We have been interested in computer graphics since its early days, and have seen all sorts of efforts towards visualising gun fire, explosions, flames and smoke. We must say that it is rare that artificial pyrotechnical effects are convincing or just resemble real fire. As they make a really good impression, however, people tend to accept them as fire. This gives us the notion that maybe it is not always necessary to have a one hundred percent perfect flame.

Simulation of fire has not yet explicitly demanded a perfect visualisation of the phenomenon, as the simulations are mostly used for examination of the spread of fire. The usual visualisation is made on areas that are so large, that a visualisation of the flames is not needed. The usual way of visualising fire is by making a map of e.g. a forest, and then mark the areas that are slowly engulfed by flames. Normally the areas are marked with a temperature indication colour, and there is therefore no need for flame visualisation. In [Bukowski and Séquin, 1997]s "VRML office fire simulation", the visualisation of fire has been reduced to simple polygons with texture maps of "fire", and a colouring of the environment's texture simulating the surrounding smoke (See figure 1.1). Bukowskis future research, however, has more focus on the look and real-time rendering, moving the development environment from the expensive Silicon Graphics equipment to the much less expensive PC.

Ed Galea from the University of Greenwich also simulates fire, and among other things he and his co-workers study the spread of heat and pressure. In a virtual room with a few large obstacles a fire is lit and allowed to evolve. To visualise this, a vertical plane bisecting the room (which is always a rectangle) is visualised on the computer in the form of a map of temperatures, indicating where the room is affected by the fire. The simulation is shown with cool blue colours fading through red to infernal white. The depiction includes isotherms and isobars. The simulation has been constructed for people wanting accuracy and legibility and therefore there is no need for fancy graphics showing the turbulent features of fire [Galea, 1997]. For more examples see figure A.5 and A.6 in appendix A.

The film industry has lately begun a comprehensive use of computer rendered effects, including fire. Since the late 70's movies—like "Star Wars"—have demanded artificial

Figure 1.1: *An example of simulated fire—from [Bukowski and Séquin, 1997].* †

special effects incurring enormous expenses and lots of problems capturing it properly due to its extreme behaviour. Since then, the special effects have become a digital realm. To avoid the tedious task of redoing explosions, compositing them onto the film (often done so in up to 15 layers), [Reeves, 1983] created an astonishing effect in "Star Trek: The Wrath of Khan" with one of the first successful attempts with particle systems (See figure 1.3). To a certain extent explosions and fires in films today are created artificially — and convincingly too. Films like "Independence Day—ID4", "La Cité des enfants perdus" and the re-release of "Star Wars" have widely used digitally enhanced fire and explosions. Digital fire is very hard to spot when it is subtle and superimposed onto real life footage as the two things

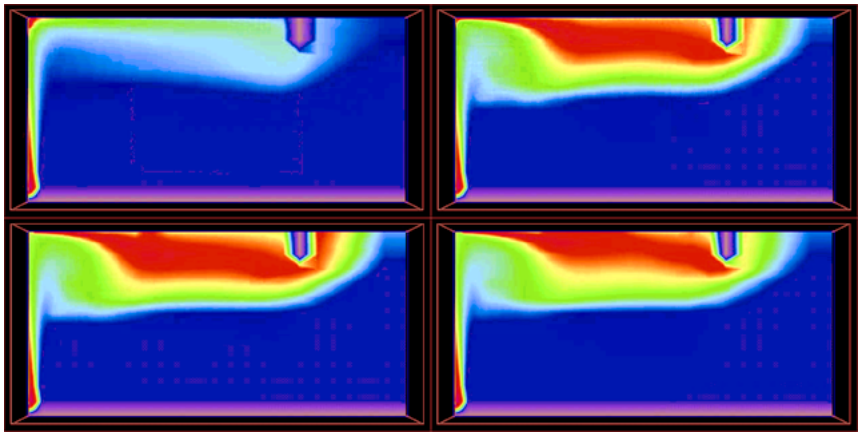


Figure 1.2: *An example of simulated fire—from [Galea, 1998]. This is four frames from an animation of a simulation of a small fire in a rectangular room with an obstacle in the ceiling. This helps to show how large an obstacle is needed to block a certain amount of heat from a fire source. The four frames are sampled at time 0.5, 2.0, 4.0 and 6.0 seconds (left to right, top to bottom). See more examples in the image appendix, figure A.5 and A.6 on page 164.* †

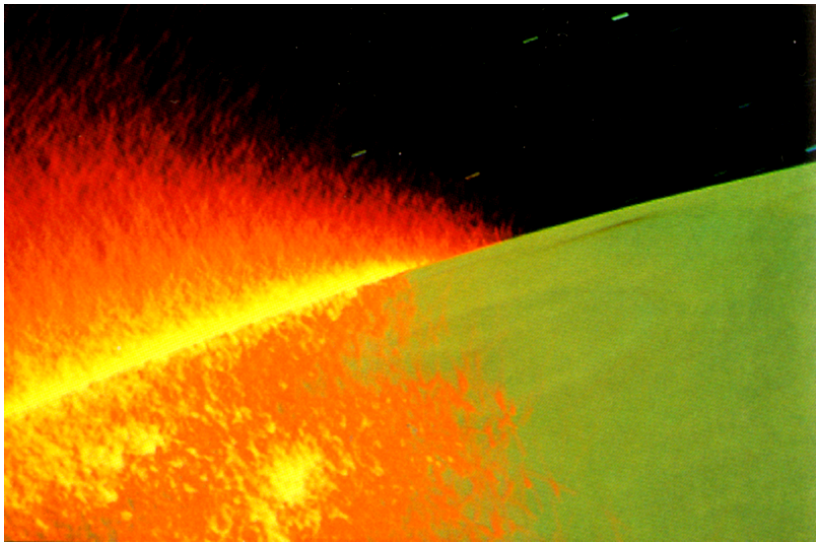


Figure 1.3: *The Genesis Bomb—from [Paramount, 1982].* †

tend to blend well. It is only when superimposed onto clean surfaces—as surfaces are in hand drawn animated pictures—that the fire looks artificial or awkward due to its large level of detail. Quite evident examples of this are "The Hunch back of Notre Dame" from Walt Disney Corp. and "Anastasia" from Don Bluth Pictures—both feature 2D and 3D computer generated fire that is very distinct from the also widely used traditionally hand animated (two dimensional) fire. Lately the 3D CGI Pixar/Disney feature "A Bug's Life" shows a fantastic real-life looking fire, but to the trained eye—we can tell—it still seems like a two dimensional forgery.

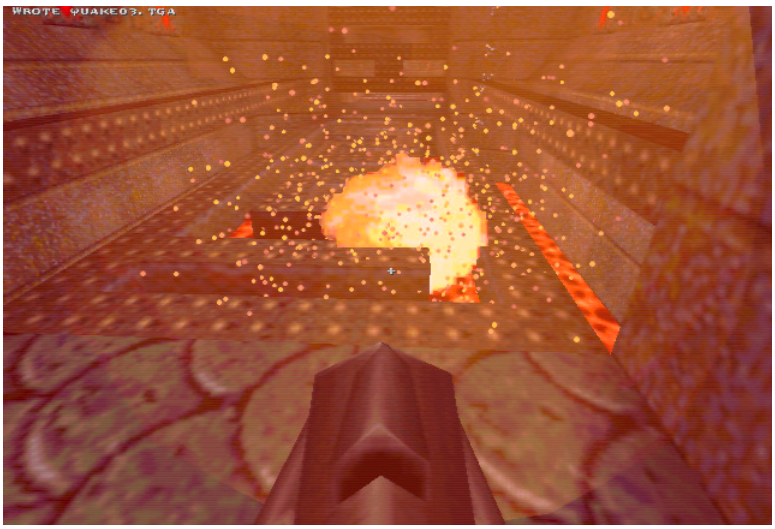


Figure 1.4: *An example of fire from the computer game Quake[Carmack et al., 1996]. Notice that it is a hybrid of three components: 1) prerendered bitmap fire (the core), 2) a real time rendered glow (transparent spheres) and 3) the real time animated and rendered, glowing particles. The explosions work very well while playing the game, this snapshot, however, is another story.* †

Computer games have had explosions and fire since their early days. The effects fall into two categories: Prerendered fire and real-time rendered fire. The first category uses rendering techniques that can be of arbitrary complexity (e.g. hand drawn animation) since the in-game rendering is merely bitmap pasting. This has been done to avoid the relative enormous rendering time—but the result is static and therefore loses its attractiveness quite fast (it becomes better, though, when altered by real-time effects, like bitmap warping). The second category has just recently become feasible as the hardware's rendering speed has increased considerably, adding more and more detailed special effects.

Lately games like Quake (figure 1.4) and Unreal feature quite stunning results, since a wide variety of lighting and rendering techniques have been used. Even though it once again only resembles fire, it is evident that real-time computer animated fire may not be impossible to obtain.

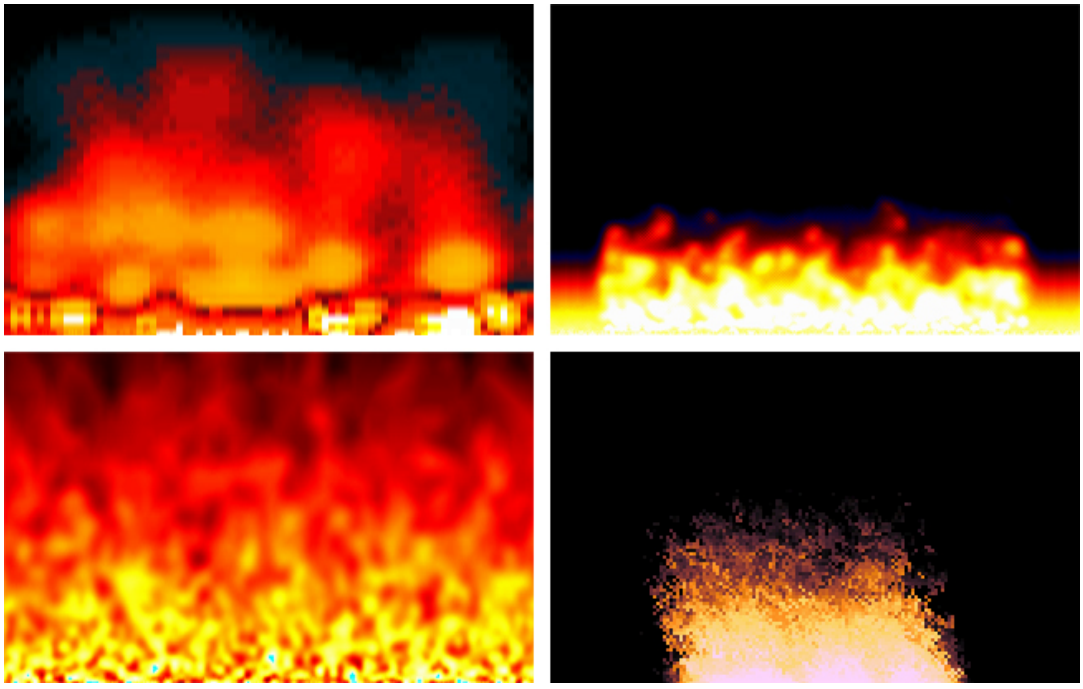


Figure 1.5: *Four pictures of demo fire which are done almost only by averaging (see section 4.1 for a detailed explanation). Notice the large difference in the implementations (hence the look of the fire) ranging from crude averaging to the more advanced averaging including calculation of wind, spread, temperature and sudden fuel bursts. All the demos run at 25-75 frames per second with 320x240 or 640x480 pixels in 8 bit colour depth (256 colours). The demos are made by (left-right-top-bottom) 1) Juan Carlos Arévalo Baeza, 2) gaffer@zip.com.au, 3) Bernard Schmitz & Christopher L. Tumber and 4) Frank Jan Sørensen.* †

Since 1993 people from the *"demo scene"*[1] have tried to do real-time rendered fire, and it has actually been done in only 128 bytes, running 320 by 240 pixels at least 25 frames per second on an average Intel Pentium based computer (see figure 1.5). The results are really nice and even though the underlying models have nothing to do with real fire and only resemble fire in a very simple way, we will discuss them in section 4.1, also because they have been a source of inspiration.

These promising attempts of visualisation and usage still lack details. What is sought—as with most computer graphics—is to make a depiction that looks like the real thing, a depiction that tricks the mind in such a way that the mind never suspects that the visualisation is artificial. Our motivation is easy to derive from that statement: We would like to contribute to the enhancement of computer based fire visualisation.

1.2 Overview of this thesis

When writing a thesis like this, it is important for the author to ensure that the reader has a general idea at all times of *what* he or she is reading about in the particular part and *why*. This should be done to avoid that any part of the thesis is misunderstood or interpreted incorrectly.

To overcome the paradox of already needing to know what you are confronted with for (perhaps) the first time, we have decided to split this thesis up in parts according to figure 1.6. The figure's purpose is to visualise a division of the topics in adjacent parts of computer graphics.

We first distinguish between two and three dimensional fire. The reason for this is that there is a large difference in how they are modelled, animated and visualised. The choice depends on what the user would like to have, but it is mostly dictated from the sort of application in mind. The use of three-dimensional (or volumetric) fire is rare, as fast real-time rendering is on the top of all wish lists for Christmas. The use of two dimensional rendering in a three dimensional environment is common due to its low complexity, as we will show later.

After this bisection a discussion concerning modelling is in place. The choice of a model almost implies the way the model is animated. There is only one modelling method which can be animated in two or more quite different ways, namely the particle systems (see figure 1.6).

When the modelling and animation methods have been decided the next thing to do is to visualise the animated model. Usually, there are several ways of visualising each model and the choice of rendering method should depend on the level of demanded speed and detail.

Figure 1.6 should therefore be read as follows: A path from the root to a leaf results in a distinct way of modelling, animating and rendering fire. The result depends on choices between speed and level of detail, as in so many computer graphics issues. Having this in mind this thesis has been divided into the following chapters:

[1] By "demo scene" we refer to people creating a certain kind of programs often released at "demo parties". These programs consist of a mixture of programming, sound, and graphics, are real-time applications, and are written with only three purposes: to boast, to entertain, and to achieve recognition by the other demo programmers. Demos are always aiming for doing something even faster, fancier, more "cool". One of the attempts is to visualise fire. For further information, please have a look at: http://www.scene.org.

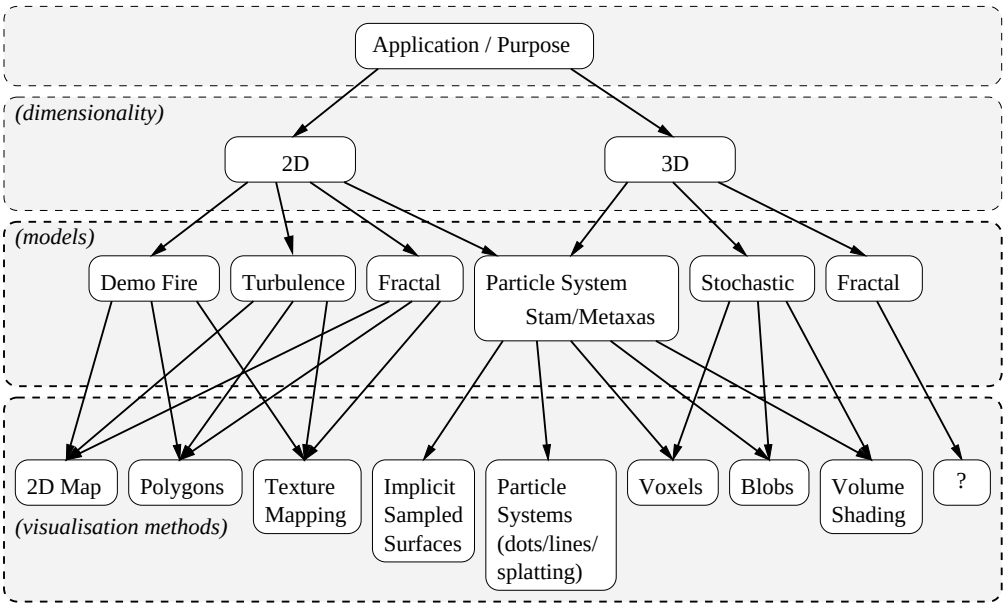


Figure 1.6: *Different methods for modelling and visualising fire, and how they could be related in an implementation.*

1 Introduction This chapter is the introductory chapter. You are reading it right now.

2 Considerations, p. 13. When starting on a large project with a general approach like this, discussions are needed on issues not directly related but still relevant. We have dealt with the level of detail combined with level of speed, and finally a section about related topics concerning the computer graphics camera model.

3 Fire and Smoke, p. 20. What is fire and smoke? The common comprehension of fire and smoke is not correct. In this chapter we try to unveil the visual characteristics of fire and smoke. These phenomena are very complex, involving huge amounts of chemistry and physics. Nevertheless, we have made an effort to explain them using words and images instead of formulas and equations.

4 Modelling & animation of fire in 2D, p. 35. There is a first time for everything. The two dimensional fire is a very good place to start out as it is simple to comprehend and since the force of the available models is that they are fast. This section shows the most common first attempts on making fire.

5 Modelling & animation of fire in 3D, p. 44. We must model and animate in three dimensions if we want a visualisation with a more real-life feeling. In this chapter, we discuss several methods of modelling and animating fire and smoke, including pros and cons, as we discuss many of the tricks used today in computer graphics.

6 Visualisation, p. 77. Visualisation—in particular of fire—can be done in many ways, each with its own look and feel. The variety of rendering techniques makes it possible to represent the model in several levels of complexity and detail, which (for instance)

might enable visualisations to be previewed in real time before pressing the button on the computer that makes it go silent for weeks and weeks.

7 Implementation, p. 101. This chapter concerns our implementation. This is a combination of what we believe is the best of the earlier chapters' theory. We present an overview of our program to ease a possible future implementation. We have tried to limit the amount of specific implementation details (the chapter does not include C++ code), as most features are dealt with in the theory chapters. It does include an overview of our custom made raytracer, the file format for making images and animations and some very simple examples of these, a list of the most important switches for the raytracer, how the particle systems work, how the radiosity class is made, and finally how these features are handled by the raytracing algorithm.

8 Results, p. 115. Our efforts end up in a lot of results. We test the theory discussed in earlier chapters, and describe the results. This way the chapter can work as a quick overview of the more interesting areas covered by this thesis. We have made example renderings of the parameters directly concerning the result of the visualisation: radiosity, blobs, shadowing, motion fields, turbulence, animations, a comparison of rendering quality and rendering speed, and finally an approximation of scattering.

9 Summary & conclusion, p. 146. For the final round up, we have made a conclusion. It deals with the goals we have set: our results, our expectations, the availability of digital fire, suggestions for future research and related literature.

Appendix A Additional images. This appendix contains images not presented within the text. The images presented here are either images that just had to be shown in colour, images we could not find a proper place for in the text, or images showing the general capabilities of our raytracer.

Appendix B Enlarged images. The images presented here are enlarged colour versions of images from the text. This will enable a closer examination of the details in the pictures.

Appendix C The TSR raytracer. This appendix contains a short description of the scene file format of the raytracer, as well as the collection of scene files used to create the examples of chapter 8. TSR is an abbreviation of "Theo's & Søren's Raytracer".

Appendix D Program listings. Contains program listings. The programs are: "Cellular automated fire", "Turbulence advected fire shape", "Bitmap pasting", and "the TSR Raytracer".

1.3 The enclosed CD

For those reading the first print of this thesis, we have enclosed a CD-ROM containing all of the pictures presented in the figures and the developed applications, all of which are executable under Windows 9x and NT. The TSR raytracer is also available under HP-UX 10. The file named `readme.txt` in the root directory contains a full description of the CD-ROM.

The applications

We have written several applications which all produce picture files as output. The file format is the TARGA image file format which is supported by most popular computer graphics image viewers. The TARGA file format is extremely simple, supports 8, 24 and 32 bit run-length encoded images, but it does take up a lot of disk storage since it is lossless. A complete specification of the TARGA file format(s) can be found in [Murray and van Ryper, 1994].

The programs that are executable under Windows 9X and NT are placed in the directory named `Windows9x`, the raytracer for HP-UX 10 is placed in `HP-UX10`. *Note:* If the applications are run directly from the CD-ROM, errors might occur since the output files they produce cannot be written to the CD-ROM.

The images

The printing techniques available today leave much to wish for, so we have also enclosed all the the images presented in this thesis. You are then able to look at details and get the real colour impression. The pictures have not been gamma-corrected, since we prefer to use gamma-correcting monitors and software picture viewers.

The pictures are both in TARGA and JPEG format on the CD-ROM. They are named by figure number (figure 3.1 would then be named `figure3.1.tga` and `figure3.1.jpg`). The files are placed in the directories `PicturesTGA` and `PicturesJPEG`.

The animations

We have also enclosed the animations that we have made. They are converted into the MPEG file format, which should be viewable on all computer platforms and operating systems. These files are placed in the directory `Animations`.

Miscellaneous

Here we have gathered additional related material (animations, images, and applications). For instance referenced animations not made by us are placed here. The directory name is `Misc`.

Chapter 2

Considerations

"Every technology goes through three stages: first, a crudely simple and quite unsatisfactory gadget; second, an enormously complicated group of gadgets designed to overcome the shortcomings of the original and acheiving thereby somewhat satisfactory performance through extremely complex compromise; third, a final proper design therefrom"

—Robert A. Heinlein

The human mind is very easy to trick. It uses a combination of top-down and bottom-up parsing to make a comprehension of what it currently sees([Eysenck and Keane, 1990] and [Humphreys and Bruce, 1989]). The top-down parsing is a semantic match with what is seen, which is refined as more and more detail is confirmed. The bottom-up process is more syntax oriented, as it matches on small details, which then are combined to match on "known" more complex objects. When the mind has ruled out most of the existing ambiguity—top-down and bottom-up—the results are combined, hopefully without conflicting properties. Glitches in the correspondence between the top-down and bottom-up evaluations are not always detected and this is why we are "fooled" by trick illustrations or when we tend to pick only one out of many ways to understand an illustration. Furthermore if we believe that one of the matches is "100%" true, we might ignore the result of the other. This is most evident when the mind is exposed to trick illustrations or just the everyday icon [Engell-Nielsen et al., 1994]. Everybody who has seen a fire burning in a fireplace will never doubt that the three pictures in figure 2.1 are fires.

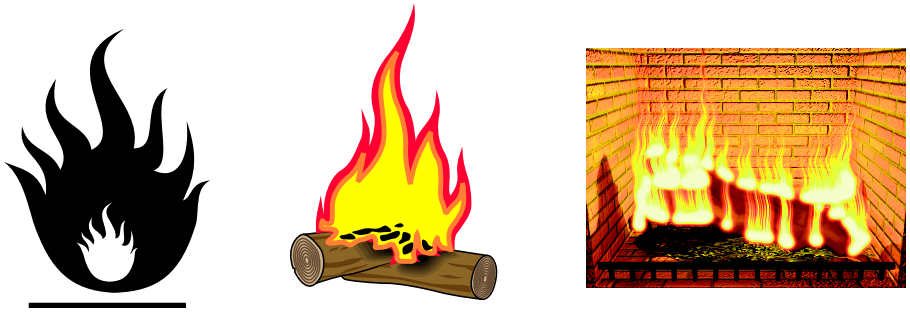


Figure 2.1: *Three simple illustrations showing fire*

This lead us to a discussion of when to use 2D fire, $2\frac{1}{2}$D fire (2D in a 3D world), or volumetric fire (true 3D); The level of detail and photo-realism has an impact on our recognition of the phenomenon.

2.1 Realism in computer animations

Since a complete emulation of fire today is virtually impossible due to the limitations of the available computational power and computer memory, we will in order to give an overview of what is feasible.

We break digital visual reproduction up into three parts:

An emulation part All that can be remade or approximated by numerical means and algorithms with no error or a parameterised minimal error. In other words the perfect recreation.

It is very lonely in this part of the computer graphics world. We cannot think of any modelling or visualisation method that is a completely accurate remake of something real-life. Take, for example the usual lighting models. They are discrete, band-limited to the red, green and blue components since a complete colour band distribution is tiresome to work with (for a discussion of this see [Clausen, 1998]). Even the most accurate model for radiosity assumes that all objects are made from perfectly even planes, an assumption that is inaccurate. Animation of solid objects comes close to how they behave in *our* world, but only as long as no interaction between objects occurs. As soon as objects collide and bounce off each other hacks, manual animation, or coarse approximations are applied.

Modelling, animation and visualisation of fire and smoke does certainly not fall in this category, since these phenomena are too complex to represent and calculate unless huge simplifications are applied, thus ending up in the next category:

A simulation part Where an emulation of a phenomenon is discarded if it is virtually impossible to remake and replaced with a pseudo-realistic look-alike effect. In other words a lossy recreation.

Most of the rendering and modelling tools fall into this category, simply because our world is too complex to model. The simplest real-world object can only be rendered correctly given a complete model of the object and its surroundings if a huge amount of time is spent. Most visualisation methods are based on *ad hoc* experiments, derived from either brilliant math deduction with vast simplifications or just late night hacks. There is historically no shame to this since all the people in the business use them and as long as the results look good everybody will be happy.

This is where modelling, animation and visualisation of fire and smoke is forced to be at the moment—and most likely will be for ever. How should these pseudo-realistic effects then be made? This is the main issue in this thesis.

Insignificance Where an effect or a detail is discarded because of one out of two reasons: *1)* some details are too small for the eye to notice and thereby too costly to reproduce whatever their costs are, small or large. *2)* some details' reproduction cost imply their insignificance. The detail is simply discarded because it will take too long to compute. This has been the case until recently with for instance radiosity and real time animated fire in games.

These are the reasons why most reproduction techniques end up in the second category, the simulation part, as the sum of barely visual details may well be a very important visual impression. The lack of knowledge on these details makes the use of hacks a very easy alternative and a good computational solution compared to a cumbersome and perhaps 100% correct model.

Fire can be computed and visualised in both two and three dimensions. Sometimes it is sufficient to use only a two dimensional fire and sometimes the rendering time of the three dimensional fire simply rules it out.

In this thesis we present several modelling, animation, and visualisation tools for 2D, $2\frac{1}{2}$D, and 3D fire. For now, we will give a very quick overview of some of the most important ones by making a simple set of comparisons of how useful *we* find the tools for creating fire, and how they perform in an implementation (we have implemented all of them).

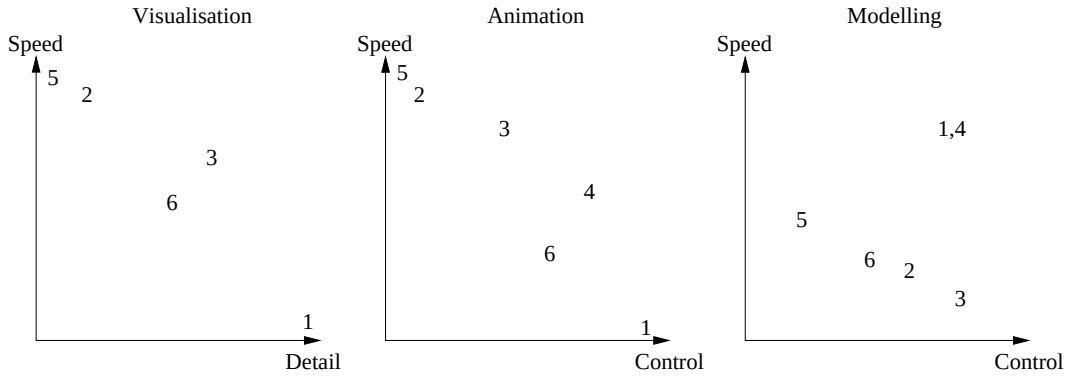


Figure 2.2: *Three indications of how different modelling, animation, and visualisation methods perform. The graphs show: visualisation rendering speed versus level of detail obtained, animation speed versus level of users control, and modelling speed versus level of user control. The axis indicates low value close to the origo, high value close to the arrows. Each method is placed where we believe it to belong relatively to the other methods. This is not an absolute index. Methods are 1) Blobs, 2) Bitmap textures, 3) Bitmap splatting, 4) Particle systems, 5) 2D demo fire, 6) 2D fire with turbulence.*

All of the techniques from figure 2.2 will be described in much more detail later, so for now we will only give a short presentation. The techniques are:

1. Blobs	A 3D visualisation tool, that requires raytracing. (Section 6.9)
2. Bitmap textures	2D pre-calculated (or pre-recorded) fire, that is pasted into a 3D scene — $2\frac{1}{2}$D. (section 6.5)
3. Bitmap splatting	A set of 2D 'fire' textures, that are inserted and possibly distorted in a 3D world — $2\frac{1}{2}$D. (section 6.8.1)
4. Particle systems	Mainly a modelling and animation tool. Is used to manage and animate other objects—for instance blobs (section 5.2). To place particle systems in the visualisation graph gives no sense.
5. 2D demo fire	Realtime rendered fire in 2D. Must be very fast and consume a minimum of memory. (section 4.1)
6. 2D turbulence	A more sophisticated 2D fire, using a noise function to give the fire a turbulent look (section 4.1.2)

In figure 2.2 visualisation speed is the rendering time for each picture, animation speed is the amount of time used to calculate each new frame in an animation, and modelling speed is the amount of time consumed by the user to create the wanted fire scene.

2.2 Computer graphics essentials

In this section we will like to direct the attention of the reader to some of the basics in computer graphics. Most are related to the camera model used in this thesis and the limitations of it. The usual camera model as we know it from [Foley et al., 1990] is very simple. Since we are in pursuit of photo realism the model is—in some cases—not general enough and some choices must be made to ensure a sufficient level of realism.

2.2.1 Motion blur

"Motion blur" is an everyday part of our lives. As the name implies it is a blurring of objects that move relatively to the observer, hence motion blur is also "caused" by a moving[1] observer. Motion blur occurs in photographs and film but it also occurs in the human perception unit, the brain.

The human brain is capable of sending a maximum of 60-90 'frames' per second (depending on the person) from the light sensitive neurons in the retina to the visual cortex in the back of the brain through two sets of interconnected neuron bundles. The 60-90 frames per second limit comes from a hard limit on time used to reset the neurons to send off another large enough action potential and the speed of neuron signals. When the received light is constant (spatially and temporally) there is no motion blur[2]. On the other hand when fluctuations occur in the received light resulting in a temporally distributed expression of light this is perceived as motion blur caused by movement. The brain then uses hard wired motion blur detection and temporal depth cuing to estimate direction and speed (but that is another story). Motion blur is a natural phenomenon in our daily life (we even take advantage of it), which is the reason why attempts are made to simulate or emulate it.

In all computer graphics systems that try to redo the moving objects of real life, the shutter speed of the camera plays a certain role in the level of photo realism. Nevertheless when frames in an animation are rendered without taking motion blur into consideration—as they are done mostly—the result is a simulation of a camera with infinite shutter speed (and thus no motion blur artifacts).

In situations where the camera (or spectator) is close to moving objects, the speed of the motion of the object projected onto the film or the retina is relatively high (compared to the speed of an object moving with the same speed but further away). Thus the motion will most probably be blurred. Even the simplest flame from a burning candle is sometimes exposed to sudden wind blows and draft which makes the flame flicker. Therefore motion blur is needed to obtain better photo realism. If an animation is done with, for example, 25 frames per second, there is 0.04 seconds between each frame in which the camera shutter is open some of time and the film inside the camera is exposed accordingly. This results in motion blur since the film is exposed to light "travelling" across the area of the picture (see figure 2.3).

Motion blur is briefly described in [Foley et al., 1990] and the more than basic technical issues must be sought there. There are two simple ways of rendering an image with this

[1] By "moving" we mean translation, rotation, scaling and deformation.

[2] The eyes never stay still as they move slightly in *microsaccades* to eliminate blindness caused by constant exposure of light of each single neuron in the retina. This has no consequence to the perception of motion blur as the microsaccades are filtered by the brain.

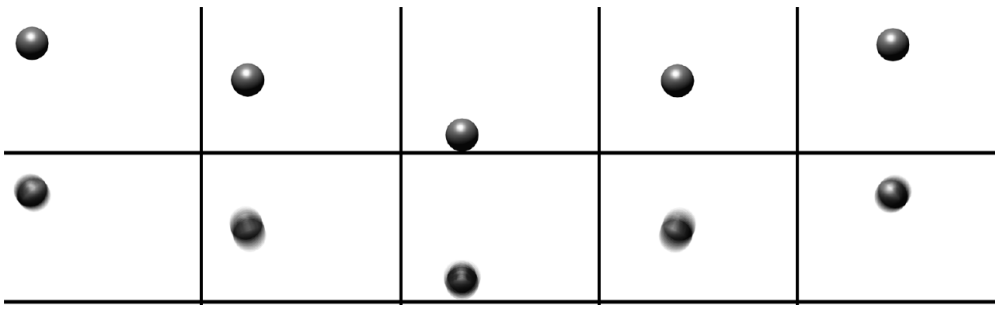


Figure 2.3: *An example of motion blur. The figure shows two visualisations of the same scene. The bottom row of pictures has been motion blurred by averaging five subsequent pictures in the animation sequence. This does not always look good in still pictures but can make a big difference in an animated sequence.*

effect, both immensely costly in rendering time and the second in memory usage. The amount of time and memory used is proportional to the desired accuracy as they are both based on an uneven number n of "renderings" centered at time t (measured in frames, of which there are usually 25 each second[3]), ranging from $t-\frac{1}{2}$ to $t+\frac{1}{2}$ (not necessarily evenly spread). The first method is quite naïve, since it takes a weighted average of the images, so the image at time t is weighed most, with the weights decreasing from t in both time directions. This is called *scene motion blur*. The second algorithm is more refined as it only renders the objects in motion, but could end up rendering the entire visible part of the world as the first method does. The method makes replicas of the objects in motion and inserts them at their positions at times distributed around t as mentioned above. To give the effect of fading, the method furthermore assigns suitable transparency values to the objects, so that at time t the object is opaque and is most transparent at time $t-\frac{1}{2}$ and $t+\frac{1}{2}$. Then the entire scene is rendered, at the cost of a severe increase in the number of primitives. This method is called *object motion blur*.

However, motion blur is what we would like to call "nice to have", but currently not essential to our project. Also, technical issues for each visualisation method will demand a specific rendering method for motion blur, and a discussion on all these would not only complicate matters discussed, but also severely increase the size of this thesis. So we move this interesting issue to the list of future projects.

2.2.2 The camera lens

A real-world camera has a lens which must be modelled and implemented if the exact same kind of images are wanted, i.e. if the computer generated images are supposed to be superimposed onto the live footage. This is a science in itself and a complicated one too. A complete remake of a camera lens is a very nice thing to have, but it is an unnecessary complication in this thesis.

Lens flares will not be discussed as they are merely another effect[4]. Nor have we dealt with prism effects which are related to lens flares.

[3]People interested in animation usable for TV know this as 50 half-frames or *fields*. Each field consists of every second image scan line. "Fields" should not be confused with the same term used in the rest of this thesis.

[4]Furthermore lens flares are (in our opinion) used too eagerly in films, mostly to cover up bad CGI.

Weird angles and field of view

Wide angle, fish eye and related lens effects are a natural part of the normal camera model. The ways of reproducing these effects are known (they can be found in [Watkins et al., 1992] and [Foley et al., 1990]). If using raytracing as a visualisation tool it is relatively easy to implement camera lenses (unless correct lens flares must be calculated), but in the case of scanline rendering this issue is indeed important.

Since we find that different lenses have no direct relevance to digital fire, we have chosen not to deal with this. Furthermore our main visualisation tools presented in section 6.9 (blobs) demand the use of raytracing, so it should be possible to add lenses later.

Depth of field

Depth of field (also dealt with in [Foley et al., 1990]) regards the focus of the camera lens, which results in a blurring of things being very distant or very close to the camera and things in the focus range will appear clear and sharp. This can be both simulated and emulated, but at high computational cost if precision and photo realism is wanted.

One of the general ideas is to oversample each pixel which makes the rendering process more costly. Another way is to blur the image by convolving a Gaussian bell curve on each pixel position according to its z-buffer depth value, back to front without affecting pixels in front.

We have—unfortunately—not had the time to indulge ourselves thoroughly with the knowledge on how to remake this lens effect, but it is important if the replication should be perfect. Depth of field is not of primary importance to this thesis. It is a possible 'nice touch' to be added at a later stage.

2.2.3 The sound of fire

We are going to deal extensively with the visual characteristics of fire and smoke. However, we will—just for the fun and interest of it—spend a brief moment discussing on the sound of fire.

It is no surprise that animations work much better if they are combined with good sound effects. The animation becomes much more alive and it is easier to relate it to the real world as the sounds give us further indications of what we are visually exposed to.

This must also concern animation of fire. But, artificial rendering of (spatial) sounds of combustion and interaction between this and animations of fire is an enormous task and we have decided not to deal with this at all. It is quite an interesting field, though, and we believe it is most likely that it has never been studied. For the time being it is easiest and cheapest to obtain the sound of flames the old fashion way: manually with a microphone near blazing flames.

2.3 Modelling, animation and visualisation

Anything that moves in three dimensions and is to be displayed by a computer has to be broken up into three different tasks: Modelling, Animation and Visualisation. Modelling concerns the process of understanding the subject that has to be rendered. Thus it has to be broken up into a finite amount of properties with—to a certain extent—discrete values

that represent the subject fully. This is a problem since we do *not* fully know what fire really is. The same problems arise in animating and visualising, and the following question emerges: "What can be done to solve this?".

First of all, we must do an analysis of fire. Inspiration for such an analysis can be found in [Landau and Lifshitz, 1995] and [Varlin, 1973].

Chapter 3

Fire & Smoke

"Ah, this is futile. Five hundred and seventy-three committee meetings, and you haven't even discovered fire *yet!"*

—Ford Prefect, The Hitchhiker's Guide to the galaxy.

The common perception of an object on fire is, that it is the object itself that burns. But it is not the object that burns, as it is the fuel (which for a piece of wood would be sap and wood oils) inside the object that is boiled to the surface, vaporised and then ignited. When asked what fire really is, people start describing its characteristics: It is yellow (mostly), really hot, it waves like steam in the air and it has the capacity of burning objects and thereby creating soot and smoke. When we asked several of our fellow students on campus to describe fire they all came up with scientific explanations based on chemistry and physics.

But what is fire made of? What is its atomic structure, what causes things to burst into flames in the first place and why can not all materials be made to produce flames? When Ed Galea and Roger Doonan both were confronted with these questions their answers were:

"Fire involves a chemical reaction between fuel and atmospheric oxygen. Once initiated it is self-sustaining, generates high temperatures and releases a combination of heat, light, noxious gasses and particulate matter. The visible flame is the region in which this chemical process occurs and so flame is essentially a gas phase phenomenon. For flaming combustion to occur, solid and liquid fuels must be converted into gaseous form. For liquid fuels this is achieved by evaporative boiling. For solid fuels, the solid is chemically decomposed through the process of pyrolysis to generate volatile gasses.", Ed Galea, University of Greenwich.

"A flame is a region containing very hot atoms. At high enough temperatures all atoms will emit energy in the form of light as their electrons, which have been prompted to higher energy levels by absorbing heat energy, fall to lower energy states. Because this light is emitted in discrete quanta according to the relationship $E = hv$ (where E =energy, h =Planck's constant and v =frequency), flame colour is related to the magnitude of the energy quantum which is transformed to light. This can most easily be seen with a Bunsen burner. A Bunsen burner that has a choked air supply burns cool, the light emissions from carbon atoms are relatively low in energy and appear more red or orange. However, when the Bunsen is allowed air so that combustion is complete, the flame is hotter and the light emitted is of a higher energy and frequency and appears blue. The luminescence of a flame is only half of the story. The structure of the flame region is important to understand too. The flame area in a normal combustion environment, such as an open-air bonfire, is structured by convection currents which form as hotter, lighter air rises and allows cooler fresh air to replace it. It is this channelling effect and movement of air that shapes the dancing flames. It is interesting that in space, in zero gravity, the hotter and cooler air cannot move by convection, so flames take on weird shapes and may be stifled by their own combustion products.", Dr. Roger C.P. Doonan, University of Bournemouth.

It is quite important to understand the concept of fire in its tiniest detail, although understanding this will not at all give a comprehension of the macroscopic features of the phenomenon. So a description of fire must be dealt with at two levels (for our purposes two suffice): one, which we prefer to call the scientific level which will—if we have a complete comprehension—make us capable of recreating fire to its finest detail. And another description level that deals with its overall effect on its surroundings and its visual characteristics, which we call the "visual level".

3.1 A scientific description of fire

For fire to be present three things must be available: oxygen, a flammable object, and high temperature. Burning is in other words an extreme *oxidation* of an object caused by the high temperature.

We are about to examine the burning characteristics of hydrocarbons. Hydrocarbons are very simple flammable structures and are found in all living material, which explains their existence in wood and fossil fuels. Examples could be Methane (CH_4) or Ethane (CH_3CH_3). The range of flammable substances is wider than this, but we think a discussion of these is irrelevant compared to the scope of this thesis. The only difference is that the heavier the molecules are, the more violent or turbulent chemical reaction is produced.

The burning process of e.g. a candle is a very complicated process. One of the vaporised and lit compounds—an acid—is oxidated like this ([Parbo, 1986]):

$$C_{17}H_{35}COOH \; + \; 11O_2 \; \rightarrow \; 9H_2O \; + \; 5CO_2 \; + 5 \, CO \; + \; 8C \; + \; 9H_2$$

This is just a little part of the oxidation process which gives just about the right impression: that if we should study the entire burning process of each single type of flame and compound, we indeed would end up with a thesis on chemistry and physics, and not computer graphics. So this is not the direction in which we will pursue fire.

Before we proceed we will go through several terms most likely to be new to the reader of this thesis. Some concerns the burning substance or item and some the flames themselves. These terms are general terms and not specific to computer generated fire. They are included here to ensure the reader's awareness of them and to emphasise what we are doing here.

Flash point The lowest temperature at which a substance will give off enough vapour to form a flammable mixture with air, so that when an ignition source is applied to the vapour a flash would travel over the hydrocarbonic surface — note: the rate with which the vapour is continuously fed from the surface does not have to withhold a burning flame).

Fire point Same as *flash point*, but the vapour has to be able to withhold a burning flame for at least five seconds. The heat span between *flash point* and *fire point* can be as low as 5 degrees Celsius.

The critical temperature is the temperature where, at atmospheric pressure, a specific material is inverted from a liquid to a gas or vapour.

Ignition temperature is the minimum temperature required to start a self-sustained combustion. Conditions which have an influence on this is: shape and size of the space

where the ignition occurs, concentration of the flammable substance, air mixture, the way the substance is ignited and the temperature of the ignition source. Ignition temperature is always an approximation due to all these variables.

Vapour density is the relative density compared to the surrounding air. Gas or vapour with a density less than 1.0 is lighter than air and will rise when released. Vapours with density greater than 1.0 will lift the air in a container upwards, since it is heavier than air.

Vapour pressure is the pressure exerted by a gas or vapour in a closed container (in all directions). Vapour with a density several times greater than 1.0 and with high vapour pressure will tend to turn into a inflammable mixture that is too rich to burn (see figure 3.1).

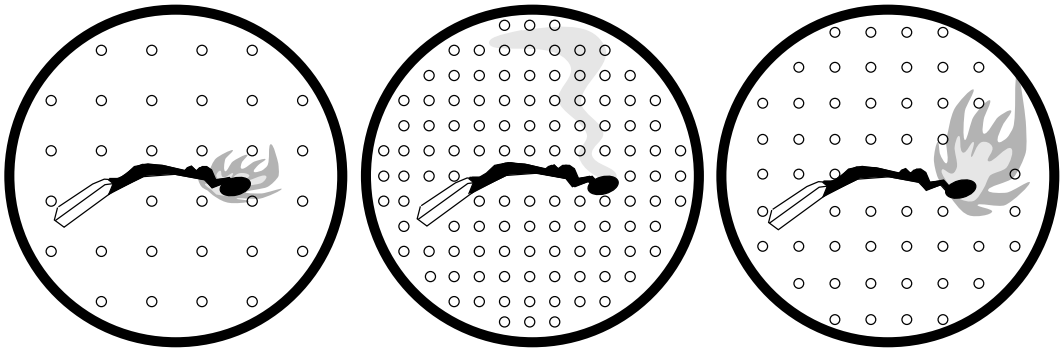


Figure 3.1: *This figure illustrates three environments with different volume-of-air ratios. The first frame shows that when there is too much space between the molecules it is impossible for the combustion to travel from one molecule to another; the mixture is too lean to burn. The middle frame shows what happens if the hydrocarbon molecules' volume percentage is larger than that of the oxygen, which leads to too little space in between hydrocarbons for letting the oxygen support combustion. The mixture is too rich to burn. The third frame shows perfect molecular air ratio. The molecules are close enough to support both flame propagation and air support for the combustion. The figure is remade from [Varlin, 1973].*

Flammable liquid is any liquid with a flash point below 93 degrees Celsius and with a vapour pressure not exceeding 40 psi (276 kPa).

Flammable vapour Consists of hydrocarbons that have left the surface of the flammable liquid because of its low flash point or because they have been boiled off the surface of the flammable liquid (vapours from flammable liquids are heavier than air expressed in terms of vapour density).

Flammable gas is a hydrocarbon molecular formation which is normally suspended in a gas. The flammable gas might be reduced to a liquid by pressure or low temperature (flammable gasses do not need to be heavier than air as with flammable vapour).

Flame propagation is the reproduction of flames through a mixture of vapour or gas with oxygen in correct volume percentage.

Combustion is when a substance is mixed with oxygen at a rate which makes it possible to evolve both light energy and heat. Flames that emerge from combustion consist of burning gasses vaporised from the combustible substance by the reaction's heat.

It is not possible to define a precise line of demarcation between "flammable" and "combustible" materials, due to the fact that a material may change behaviour when split up into tiny "dust" particles. Consider normal dust that you find on your bookshelf. If this dust is gathered and pressed together it is indeed very flammable. On the other hand if you light a match in a very (very, very) dusty room, you might end up igniting an explosion[1].

Flames are the release of energy by combustion, and combustion consumes the flame-producing substance. In other words: Flames are a chemical chain reaction in which an original ignition already has occurred. If we examine what happens when we strike a match: The source of ignition is the friction—caused by striking—which leads to the production of heat, which acts on the chemical substances within the wood of the match, causing combustion and finally flames.

Fire is vapour or gas burning in open air or in a place where air is available to form a mixture capable of burning (see figure 3.2).

Explosion is burning vapour or gas in an enclosure which, to a certain extent, can contain the pressure buildup that would cause such an enclosure to rupture violently (see figure 3.2).

Flammable gas or vapour	+	Unconfined oxygen in correct percentage	+	Source of ignition	=	Fire
Flammable vapour or gas confined so pressure builds up	+	Oxygen in correct percentage	+	Source of ignition	=	Explosion

Figure 3.2: *Conditions leading to fire and explosions*

Detonation is propagation of flames following shock waves, at very high pressure and speed. The detonation differs from a normal explosion with pressure peak values up to 20 times greater, which enables its front to travel at supersonic velocities and exert directed blows.

These related terms (picked from [Varlin, 1973]) tell us that fire behaves according to the characteristics of its surrounding environment. Fire is either fed with flammable vapour or flammable gas. Either of these must exist under conditions where temperature, vapour density and vapour pressure allows it to reach its *flash point* and withhold flame propagation.

[1] No master's thesis without small anecdotes: In Scottish distilleries it is forbidden to take photographs indoors using a camera flash due to the explosion hazard caused by the presence of ethanol and methanol fumes in the critical volume-of-air ratio!

If we dig into the chemistry literature, we find for example that *methanol* and *ethanol* have a lot of characteristics that concern combustion and flame propagation. Some of them are listed in table 3.1.

Compound name	Mass *g/mol*	Flash point *°C*	Boiling point *°C*	Auto ign. *°C*	Flam. limits *Range*	Enthalpy (c.l.) *kJ/mol*	Enthalpy (c.g.) *kJ/mol*	Enthalpy (v) *kJ/mol*
Methanol	32.04	11	65	385	6-37 %	726.1	763.7	35.21
Ethanol	46.07	12	78	363	3-19 %	1366.8	1409.4	38.56

Table 3.1: *Characteristics for Methanol and Ethanol. Flammable limits are percentage in air. Enthalpy (the amount of available energy) (c.l.) is enthalpy at combustion in liquid form, Enthalpy (c.g.) is enthalpy at combustion in gaseous form, Enthalpy (v) is enthalpy when vaporised. Note: Combustion enthalpy is negated in this table. Found in various tables in [Lide, 1992].*

3.2 Visual characteristics of fire

Fire can exist under a number of conditions of which some are relevant to this thesis and some are not. For instance fire is almost invisible in zero gravity, and furthermore flames do not have the same shape as they do on earth; they are spherical objects and produce no smoke due to its complete combustion [Christensen et al., 1998]. Since fire in zero gravity is "invisible" it will not be discussed any further.

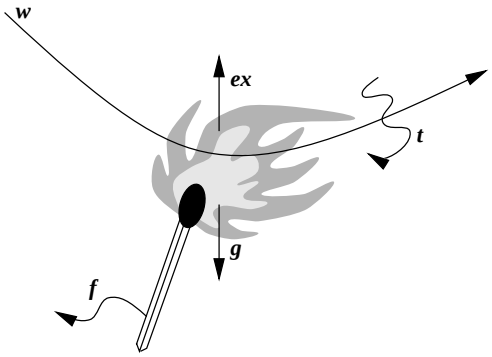


Figure 3.3: *A match and its surroundings. The visualisation of this burning match must take the internal forces and an external force into consideration. The colour of the flame indicates its temperature and the contents of compounds of the burning matter. If an object or the match itself obstructs the flame the flame should be advected accordingly. Internal forces include exothermal processes,* **ex***, and forces coming from an exploding fuel, both turbulent. External forces are* $\mathbf{F}_{ext}$*, a summation of the wind fields* **w** *including turbulence* **t***, gravity* **g** *and external motion inflicted by* **f***, which could be interpreted as friction.*

Fire can exist in different forms under different circumstances, ranging from the quietly burning candle flame to the roaring fire of a burning oil well, for instance the setup in figure 3.3, that shows some of the forces affecting the fire from a match. In the previous section we discussed how fire can be described by its physical characteristics or complexity, but it is clear that an accurate model of the physics involved in fire would be too complex as

a starting point. It is more interesting to start by describing the visual characteristics of different types of fire and then—when the visualisations are good—it might be time to look into the details of getting the fire to burn at right temperatures in the right circumstances. For now we will divide fire into groups of different visual characteristics with regards to a few physical entities:

Calm fire A good example of a calm fire is the one from a lit candle stick when it is undisturbed by sudden and violent internal and external forces such as gusts of wind. Instead it is only affected by small and slowly evolving external forces with almost no local features, which results in a flame that hardly moves. The amount of fuel is controlled and is kept at a constant rate. There is no possible way for the fire to spread but indirectly (see section 3.2.3 on page 31) due to the vertical motion of the fuel components.

This kind of fire is not a complicated one to remake because it is very regular and the fact that the flame does not interact with the burning object only simplifies the model even more. The flame is in other words easy to predict. A complete physical simulation of this kind of fire would be overkill and therefore a simple solution is preferred, unless smoke or interaction with the flame is needed.

Freely fed By a freely fed fire we think, for instance, of a flickering torch which has no lack of fuel and receives plenty of oxygen. The fire is advected by all kinds of more or less gentle external forces. This kind of fire can spread through the entire burning object, but is constrained by the upwards motion of the fuel components.

Interaction between the burning objects themselves in this kind of fire is inevitable, thus we are in need of a more complicated model.

Violent fire The example of a blazing camp fire shows the combination of large amounts of available fuel and oxygen. The fuel is vaporised freely whenever available, resulting in a somewhat irregular fuel feed. The hot fuel components generally rise vertically.

Once again interaction between the burning objects and the flames require a complicated model.

Pressure fed A blow torch or a Bunsen burner is a good example of this kind of fire. The fuel is vapour kept under pressure which means that the fuel components (when released) have velocity vectors that can be very different from those imposed by gravity, wind, and the fact that hot gaseous matter rises vertically.

Interaction between objects and the flames requires a complicated model. If object interaction is not needed, a simple model can be derived.

Explosion The average Hollywood stunt car explosion is a very complicated type of fire. One of the facts that makes this fire complicated is that the fuel is iteratively freed by shock waves, which induce new small explosions giving birth to more shock waves and so forth. Matter that is thrown about by an explosion is a mixture of three things: "dead", "inactive" and "active" objects. "Dead" objects are objects that only affect the turbulent wind flows and interact with other objects. "Inactive" objects are objects that are able to burn, i.e. either flammable objects containing fuel or unlit fuel. These types of objects also interact with all other objects. As a consequence

of this unlit fuel can turn a "dead" object into a "inactive" object. "Active" objects are lit objects or lit fuel. These objects interact with all others, causing the spread of the explosion.

This kind of fire is generally uncontrollable due to its extreme behaviour. We have not modelled this even though it *is* inarguably a kind of fire. A model of explosions deals with compressible volumes of air which would complicate the physics in this thesis considerably. Furthermore, from the start we had no intentions of modelling explosions due to the fact that they are easy to remake by mere "hacks" that resemble explosions rather well. These are incorporated into most commercial 3D computer graphics systems today (See for example *3D Studio Max* from Kinetix, *ALIAS* from Wavefront or *Softimage* from Microsoft). Nevertheless it is a very intriguing kind of fire and a very good idea for a future project.

The rest Extremes like detonating dynamite, A-bombs, or worse, we have decided to categorise as irrelevant for this thesis, since they primarily inflict pressure and shock waves and fire secondarily. We have no intentions of simulating these phenomena, but they are ideal ideas for future projects.

In this thesis we will mainly concentrate on freely fed fire, as the calm fire is "too easy" to create (and too boring to look at). Freely fed fire is one of the most usually encountered types of fire in real life, and we do not have to worry about properties of the burning material but only of the flames (as opposed to violent fire and pressure fed fire).

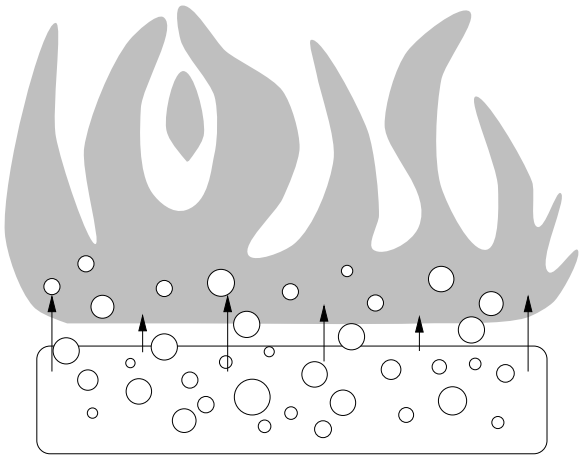


Figure 3.4: *An ideal depiction of a burning log. The fuel trapped inside is released as the surroundings are heated. In this example the fuel is released upwards, since the flames are situated in the top. If the flames engulfed the entire log, the fuel would be released correspondingly.*

Imagine the visual characteristics of a burning log, where the fuel, such as sap and wood oils, is trapped within. Figure 3.4 shows an ideal depiction of this. When the log is heated the fuel starts to move towards the heated area (the other directions are virtually impossible due to the surrounding molecules' low kinetic energy) as it begins to vaporise. As the fuel leaves the object, its molecules have high kinetic energy which makes blending with oxygen easy. As the fuel molecules come closer to the burning fire, their temperatures

rise beyond their flash point and ignition temperature where flame propagation is possible. At this point the fuel molecules start to burn or more precisely emit light because the molecules' electrons starts to do quantum leaps. Gravity and exothermal forces (ignoring turbulence caused by other forces for a moment) determine the shape of the flame, giving some of the lighter flammable fumes from the log a greater velocity than the heavier fumes. This causes turbulence resulting in the characteristic waving or flickering flame [Christensen et al., 1998].

As the molecules lose their burning capacity, they are pushed away from the centre of the heat source and the fire becomes H_2O, CO_2 and most commonly hydrogen and carbon, which visually results in smoke (soot[2] and H_2O). Smoke will be discussed in section 3.3.

3.2.1 The colours of fire

The colour of a flame is—to put it in nuclear physics' terms—produced by quantum leaps of the electrons in the compounds of the fuel. When an atom's electrons receive a large amount of kinetic energy (are hit by other particles, i.e. other electrons or a photons with great velocity) they may jump from their normal orbital to another orbital, where they stay until they release a photon with a certain level of energy (and frequency) and subsequently return. Some of these photons' frequency lie in the visible spectrum of electromagnetic radiation, which is what we call "light". Figure 3.5 shows a picture of a candlelight flame which demonstrates the range of the emanated colours.

Figure 3.5: *A candlelight flame. In the corresponding colour version in appendix B: Notice that the flame is bright yellow (almost white) in the upper regions and dark in lower part. Sometimes a fading towards blue close to the wick can be seen. Also notice how the flame actually casts a shadow on the back wall (caused by the camera blitz).* †

[2] consisting chiefly of carbon.

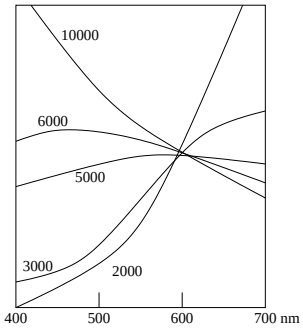


Figure 3.6: *Colour temperature is measured in Kelvin. The temperature of fire lies in the range of 1500-2000K. The black body radiation dictates thereby the black body palette (see appendix A.1).*

Usually, in the natural sciences, fire is treated ideally as a black body radiator. A *black body radiator* is an object which radiates a certain distribution of electromagnetic radiation. The characteristics of this distribution depend on the colour temperature of the object which is measured in Kelvin. This temperature is an abstract term, though, as the temperature is above what physically is possible to measure. Figure 3.6 shows several distributions at different Kelvin temperature. Since the average fire has a temperature ranging from 1500 to 2000 Kelvin, the black body radiator's distribution is easily derivable. Briefly put, a black body radiates photons in colours ranging from black to cherry red, through orange and yellow, to white (see figure A.1 in appendix A), which can be represented by the palette referred to as the "*black body palette*". A more informal explanation of this is to think of the colours that radiate from a piece of iron warmed in a forge. First it is black, then it glows red, then orange, and finally white. This black body radiator colour model is very simple but it works good for most of the models dealt with in this thesis. In real life, though, the phenomenon as such does not exist, since the amount of "yellow photons" is larger than for any other colour [Ebert et al., 1998].

Mainly there are two reasons that the flame has colours. First, the temperature of the flame, and second, the chemical structure of the flame (and the smoke rising from it). The latter is affected by incomplete combustion.

The temperature of the flame can be lowered or raised relatively by removing or feeding the flame with fuel and air. Think again of the Bunsen burner, where the flame visibly changes colour as the temperature changes.

How the temperature affect the colour of a flame can be seen by taking a long look at a lit candle. You will then realise that the part of the flame closest to the wick is invisible, which gradually changes (perpendicular to the flame's direction of combustion) to a bluish colour and then invisible. From the wick and upwards the flame is bluish or invisible, then white, fades to bright yellow for almost the entire length of the flame and then to red, then invisible. Normally no bluish colour is seen. The perpendicular blue colour is due to molecules that are lit and cooled immediately, resulting in emission of photons at around 450 nm. Upwards all sorts of photons are emitted close to the wick, and as the molecules are lit further and further away (due to lack of oxygen) they will be cooler, emitting photons with less energy, first some at 540 nm, then at 580 nm and finally at 675 nm, giving the above mentioned colour pattern.

In figure 3.5 the bluish colour is not seen, although it was present to the naked eye

Substance	Oxidising flame	Reducing flame
Chromium	Green (c.)	Green
Copper	Green (h.), blue (c.)	Red (c.), opaque (s.s.), colourless (h.)
Iron	Yellow or brown/red (h., n.s.)	Green (s.s.)
Manganese	Violet (h.c.)	Colourless (h.c.)

Table 3.2: *Compounds and their flame colours. The used abbreviations are: s.s. is super saturated, n.s. is non saturated, h. is hot, c. is cold. See section 3.2.2 for an explanation of oxidising and reducing flames.*

Frequency (THz)	Wavelength (nm)	Colour
> 750	< 400	Ultra Violet (UV)
750	400—424	Violet
707	424—491	Blue
611	491—575	Green
522	575—585	Yellow
513	585—647	Orange
428	647—700	Red
< 428	> 700	Infra Red

Table 3.3: *The colour spectrum*

when the image was taken. Notice however, that the flame actually casts a shadow on the back wall. This is caused by the light from the blitz being occluded by byproducts of incomplete combustion in the flame. This gives us an affirmative answer to a question that has been nagging us for a long time: If you point a flashlight at a bonfire, will the flames cast a shadow?

The other main reason for colouring of a flame is incomplete combustion, which is the result of burning a compound that is not split up into H_2O and CO_2 alone. Carbon and hydrogen can be seen as the remains in the above mentioned example. A compound can consist of all sorts of atoms or semi-molecules, that will change the emission energy level of some of the photons and thus make the flame glow in another colour. This could be different kinds of metals, alloys, or plastics. Table 3.2 shows a number of examples of different metals that give the flame different colour. This is related to the wavelength of the photons that are emitted from the electron quantum leaps, see table 3.3 for a reference to their wavelength.

3.2.2 Oxygen

A very important factor concerning fire is the amount of oxygen in the surrounding air. Oxygen is usually present in finite amounts. Without it, there is *no* fire. Thus oxygen is a constraint which must be considered when fire is simulated. A peculiar thing about oxygen is that if it is applied increasingly to a flame or fire, for instance a candle, it will make the flame rise and at a given point the flame will be blown out. The reason for this is a sudden decrease in available vaporised fuel because it is blown away and perhaps also due to a lowered temperature of the flame and the fuel. One perhaps known and feared thing

is "backdraft" that occurs when only enough oxygen is present to keep a microscopical fire alive on the surface of the fuel. When oxygen or air becomes available or mixed with the fuel (e.g. by the opening of a door), the entire fuel is lit instantly resulting in an explosive fire.

A flame can be in three stages. If the flame is in its *oxidising stage*, increased oxygen supply makes the flame rise. The fire is in other words capable of using even more of the available oxygen. When it is in its *reducing stage*, it is not capable of using all the available oxygen. Further supply of oxygen will not result in more fire; indeed a reduced flame may result. The latter stage is the point between an oxidising and a reducing flame, where the amount of available oxygen matches the capacity of the lit fuel, in other words: perfect volume-of-air ratio.

The colour of the flame changes accordingly to the amount of available oxygen. The reason for this is somewhat obvious as the electrons are able to "produce" photons of other frequencies. Examples of some compounds' colours in the reducing and oxidising stages can be found in table 3.2.

These effects of rising and reducing flames due to the amount of available oxygen in the air are important if a complete simulation is wanted. However these effects are quite subtle and complicated ones to remake.

3.2.3 The spread of fire

We term movement or evolution of fire *the spread of fire*. The main reason for this movement is the huge emission of heat, which leads to a heating of the surrounding areas resulting in an increased amount of released flammable vapour, which is then ignited. This can be applied to both the nearby surroundings (matter relatively close to what already is on fire, e.g. a burning log) or objects that are only in touch with or near the flame (e.g. a stick held into a fire, or a log placed near another burning log).

There are several things to take into account. Fire that "intersects" with a flammable vapour at its flash or fire point, will ignite the vapour and thereby spread. An immediate fast increase in temperature of the surroundings of the gas/vapour is expected. We call this *direct* spread of fire. An object which is inserted into or is kept near a burning fire, is bound to increase in temperature adaptively. When exposed for a period of time it may reach a temperature at or above its fire point and catch on fire. This is what we will refer to as *indirect* spread of fire.

A very special kind of indirect spread of fire is when an object that is standing close to a fire suddenly, and without any warning, bursts into flames all over. This effect, known as a *flash over*, happens because the air is heated to extreme temperatures, causing non-burning objects to heat up. If the heating is relatively uniform in the area, a non-burning object might reach its flash point on the whole surface at the same time, resulting in the above mentioned effect.

[Ahmed et al., 1994] give a very detailed description of the chemical processes involved in the spread of fire. Their work relies heavily on chemistry, and is furthermore restricted to vertical and horizontal surfaces. We find that a detailed description is outside the scope of this thesis. Instead we calculate a simple approximation to the temperature throughout the scene (section 5.5.1), and use this to see if any objects will ignite at their given positions.

3.3 Visual characteristics of smoke

As mentioned in section 3.2, incomplete combustion results in the creation of visible byproducts such as soot. When we talk of smoke, we refer to volumes of hot gas containing large amounts of those byproducts as well as tiny fragments of the burning material.

When defining the visual characteristics of smoke it is important both to capture the general appearance as well as the motion. To start with the first characteristic: The colour of smoke from a fire will almost always be very dark, because of the high levels of carbon the gas carries. But other colours might appear from the fragments of the burning material or other byproducts. For instance steam (defining steam as consisting mainly of water and no carbon) would by many people be recognised as white smoke.

3.3.1 The colours of smoke

The amount of colouring particles in the smoke can lead to very different visual characteristics, ranging from thick and completely black to thin and almost transparent. This semi-translucency makes the smoke able to cast shadows on the surroundings and even on itself. A volume of smoke can be lighted from one side without any of the light reaching through to the other side. In this thesis we call this the *self shadowing* ability of smoke.

The thickness is also a parameter. Think of a burning car tire and a lit cigarette, which produce two very different kinds of smoke. Depending on the fuel, a column of smoke will rise from the burning object. Where the smoke from the cigarette is easily advected by small currents of air, the smoke from the car tire, on the other hand, seems much more heavy and is not as easily advected as it contains huge amounts of heavy byproducts. A thick volume tends to keep together, whereas light smoke is quickly dispersed (think again of the lit cigarette).

How smoke is perceived also greatly depends on how it interacts with light. Here the following subjects must be treated.

Scattering Scattering is the effect of energy bouncing off particles (being water, dust, carbon, or other particles) that have received this energy and simply reflect all or part of it (figure 3.7). The concept itself is easy to grasp, but the computational overhead is enormous if the energy transportation inside the medium is to be calculated exactly.

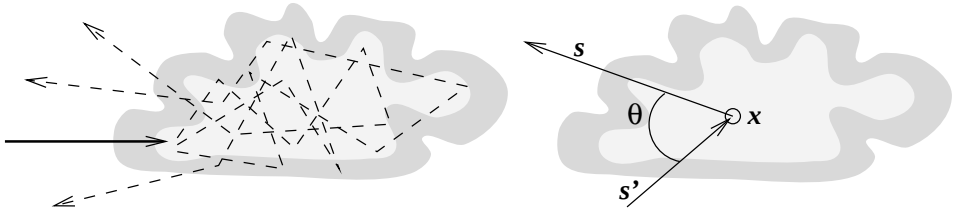


Figure 3.7: *The difference between albedo and scattering: The entering light beams are* scattered *around inside the medium until—at some point—they are either absorbed and result in a heating of the object or leave the object. The* albedo *of a medium is the total fraction of incoming light that leaves the medium again.*

The computer graphics literature usually operates with two kinds of scattering. *Single scattering* is when a light ray is only refracted once inside the medium, thus simplifying the calculations greatly. *Multiple scattering* or *self-scattering* takes place when particles emit some or all of the energy that they receive and pass it on, perhaps to other particles, and these particles are in turn able to pass the energy further on. Think of a small cloud struck by a light beam. The entering area's beam shape is quite unchanged, but as the light passes through the cloud it spreads within making the entire cloud glow on the back of it (when viewed from the light source).

To render a gaseous phenomenon with scattering, the equations of *radiative transfer* (the *scattering equations*) should be solved, as they are in [Lenoble, 1985] and [Stam, 1991]. This includes finding the amount of *radiance*, *absorption* and *albedo*, since internal and external scattering must be taken in consideration. As this requires solving a set of differential equations the computational cost is huge, which is the reason why we have chosen not to include scattering in our implementation. Furthermore scattering is not very noticeable in effects such as smoke or fire, but is more prominent in steam and clouds where the albedo is much higher. In [Rushmeier and Torrance, 1987] a way of calculating scattering in rectangular box-shaped volumes of air is presented. This has inspired us to create our own approximation of the scattering equation. This approximation as well as the algorithm proposed by [Rushmeier and Torrance, 1987] is described in section 8.8.

Albedo Albedo is the amount of incoming energy that emanates from a location on a medium. When light enters the medium it is scattered around inside, and while some of it is absorbed, a fraction of the light might manage to escape. This fraction is the albedo of the medium. A *low albedo approximation* can be done by asserting that scattering behaves with only a single scattering, which was shown—with good results—in [Kajiya and von Herzen, 1984]. When finding of the albedo it is assumed that scattering is isotropic[3] to simplify the problem.

As the albedo is so closely connected to the scattering, a good approximation to the scattering equation will at the same time result in a good albedo approximation. For further discussion on this subject refer to section 8.8.

Self shadowing When a gaseous phenomenon is illuminated by a light source it becomes brighter on the surface and inside of it. From the entry point and further through the energy is absorbed and scattered in such a way that the light does not necessarily pass through to change the illumination on the opposite side. The absorption is done according to the medium's density as light passes through it. This is called the *self shadowing* effect.

Self shadowing smoke cannot be rendered correctly without using multiple scattering. According to Kajiya and Stam the result becomes dull, even when the above mentioned low albedo approximation is used. However, we have obtained good results without using multiple scattering. This will be discussed in chapter 8, where we present our results.

[3]Isotropic: exhibiting properties with the same values when measured along axes in all directions.

3.3.2 The motion of smoke

The motion and shape of smoke are other distinct characteristics, but to describe the shape of smoke is certainly as difficult as describing the shape of fire. This is especially the case if the description should cover all kinds of smoke such as the smoke from a lit cigarette or the smoke from a roaring fire. An important feature for recognising this type of phenomenon is the motion of smoke.

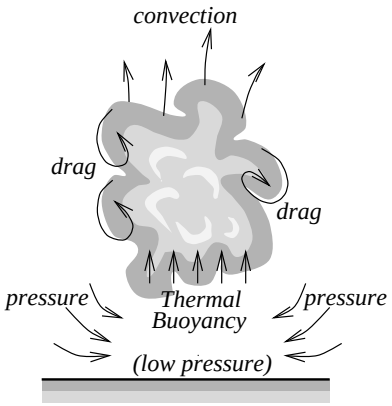


Figure 3.8: *The motion of a hot turbulent gas. The figure shows which effect contributes to which motion indicators when a hot gas interacts with a cooler environment. This figure is repeated in section 5.5.1 (figure 5.10)*

When asked to define smoke a fellow student said: "Smoke is like clouds with a purpose". The 'purpose' being that smoke tends to rise up as the phenomenon is much hotter than the surrounding air and therefore lighter. In fact, a lot of the turbulent motion that gives smoke this "hard to describe" appearance is a result of the hot gas of the smoke meeting the cold and still air of the surroundings. Figure 3.8 shows the involved effects: *drag*, *convection* , *thermal buoyancy*, and *pressure*. A deeper description of the forces involved when a hot gas rises can be found in section 5.5.1. Simple ways of creating a turbulent appearance can be found in sections 4.1.2 and 5.4.

Chapter 4

Modelling & animation of fire in 2D

"Look at arson — I mean, how many of us can honestly say, that at one point or another he hasn't set fire to some great public building? I know I have!"

—The Amazing Kargol, Monty Python

In computer graphics the tradeoff most commonly encountered is the choice between speed and realism, since high resolution super realism is not always wanted if the price is an extreme computational cost. Especially in real-time applications rendering speed has high priority. The name of the game is "bottle-neck removal"—the cost is lower quality, the benefit is fewer clock cycles used. Such speed-ups are usually gained by deploying hacks that have no relation to "real-world" observations but still do the trick. One example could be to replace an implementation using reals with one using approximated integers or fixed-point math. This could result in a chunky or colour banded visualisation, »but if it is fast enough nobody will notice the difference«. In this section we will discuss two different kinds of "hacks" that have evolved like mentioned above. Let us indulge ourselves:

4.1 Our first attempt at making fire

In computer demos (described in section 1.1) the object has been to create the fastest and smallest executable real-time renderer of fire (also briefly described in section 1.1). One of the better algorithms gives a real-time animated 'fire' as the one seen in figure 4.1. The

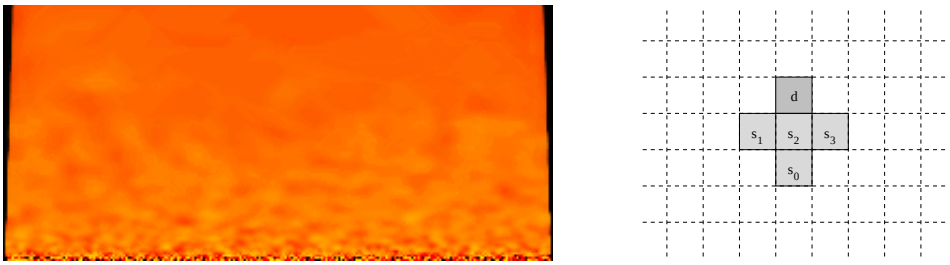


Figure 4.1: *Demo fire. Fast but not very fire-like. The image is made iteratively by averaging the pixels from top to bottom,* $d = \frac{s_0+s_1+s_2+s_3}{4}$*. New fire is inserted as random pixel values at the bottom pixel row, ranging from 0 to 255 (one byte per pixel) and a black body palette is used (see appendix A.1 on page 162 for this palette).* †

algorithm for this fire is very simple indeed, and originates from the theory of *cellular automata*. Before describing the algorithm, simply think of a grey scale image with values ranging from 0 to a maximum value, i.e. 255. The pixel values are used as an index into a colour table, namely "the black body palette" (figure A.1 in appendix A). Back to the algorithm: For each screen refresh, a row of random intensity pixels are written in the bottom of the image canvas. The whole screen is then rendered from top to bottom using a simple video feedback method: The current pixel's value, d, is found as the mean value of the pixels s_1, s_2 and s_3 in the row below, and the pixel s_0 right under but two rows below (this mean is just an example, it can be simpler or more complex). The choice of

interpolation method and mean sampling has high impact on how the fire evolve over time. The advection is only vertical and equal to the height of one pixel, which also has a great impact on its "look and feel".

4.1.1 2D fire extensions

To make the demo fire evolve in more interesting shapes and in a more controlled way two things can be done, namely changing the way the new fire is inserted and how it is removed again.

Insertion of fire

The way the new fire (or fuel) is inserted is obviously important to the shape of the fire. Even more obvious is it that a better way than inserting random pixel values at the bottom pixel row exists. If the insertion fire is animated by a function that changes through time and even if the insertion is not only done on the bottom pixel row, the result will become much more fire like. For instance, features like sparks and flame tongue bursts are easy to obtain using this rendering method. Also burning captions and logos are possible to make at low computational cost and in real-time.

Cooling map

To obtain even greater control of the fire, a cooling map is introduced. The cooling map is simply an image map that is subtracted from the fire image map to make it fade towards the background colour. This enables smooth fire along the border of the image. Logos and captions can also be enhanced with this feature. The cooling map can—as with the insertion of fire—also be animated. Figure 4.2 shows a single frame from an animated cooling map.

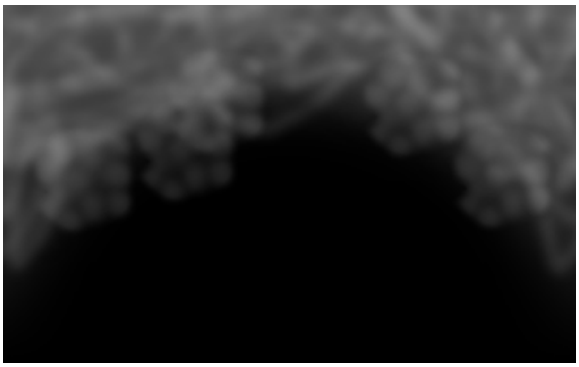


Figure 4.2: *2D animated cooling map*

2D advection

The problem with the above mentioned method is that it is very static and in the long run very dull looking. What is really needed is a better advection of the fire. We will discuss

a more advanced video feedback method and a turbulence shape warping method used on a static bitmap as described in [Watt and Watt, 1992].

Advanced video feed back

As described above video feed back is used as an iterative way of shaping the flames. This feature is expanded from a vertical only global advection, to a local all direction advection (as suggested by [Elias, 1998]). Consider for instance an image covered by a 4 by 4 grid described by a set of vertices, G. Then consider G', the set G with the vertices slightly advected by a function α. (figure 4.3).

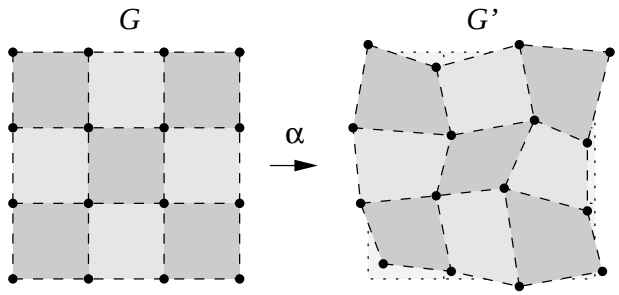


Figure 4.3: *2D grid advection. A set of vertices G are advected by a function α. This advection is used to warp an image. In this figure a checkerboard is warped through α.*

The function α advects vertices so that no grid edges will intersect and can either be static or vary over time, making the advection more "alive". This could be a simple sine and/or cosine:

$$\alpha(g_{ij}, t, size) = (\ g_{ij,x} + size\cos(T_{i,j}(t))\ ,\ g_{ij,y} + size\sin(T_{i,j}(t))\ + y_{adv}),\ g_{ij} \in G$$

where $T_{i,j}(t)$ (t being time) changes according to the position on the map if an adaptive advection is intended. y_{adv} is advection in the vertical direction. This is used to force an upwards motion to the advection. Figure 4.4 shows the surprisingly good result obtained by just applying random advection to the grid vertices.

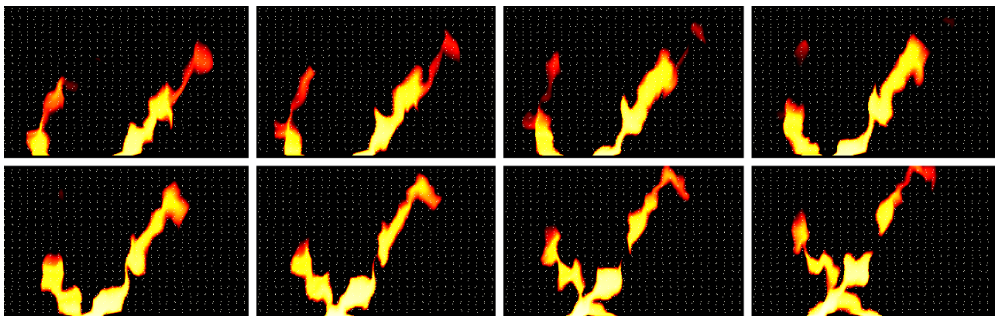


Figure 4.4: *Applying video feed back. Two small emitters of fire are moved back and forth at the bottom of the canvas. The emitted fire is then advected by video feed back. The images are the first eight images from every fifth grid advection. The small dots behind the flames show the grids vertices G and G'.*

What is needed now is a function μ that maps the image from grid G to grid G'. How the μ-mapping is done will not be discussed here since this is a matter of mere bitmap morphing, even though the way it is done affects the output (numerical accuracy, sampling methods and antialising taken into account). The result of all these changes are quite evident. The fire is now very much alive; it is no longer a homogeneous matter of yellow and red spurts that spawn in the bottom of the image and fade into the top pixel row's blackness.

4.1.2 Turbulence shape warping

A perhaps more realistic looking 2D fire can be achieved by creating a standard fire shape, and then distort it by applying shape warping [Watt and Watt, 1992]. This standard fire shape should depict the 'perfect' flame—a flame which is undisturbed by external forces such as wind or turbulence. Figure 4.5 shows such a flame. It has been created by a simple function I, which returns a value in the range $[0, 1]$ for a given (x, y) pixel coordinate.

$$I(x, y) = (1 - \frac{y}{yres}) \sin(\frac{x}{xres}\pi), \tag{4.1}$$

where $xres \times yres$ is the resolution of the image. This value is then translated to a RGB colour value using a black body palette.

Now the shape of the perfect flame is warped by adding some kind of noise to the sampling points (the x and y values of equation 4.1). Just adding random values to the sampling points will not produce the desired effect, but only reduce the quality of the image as there are no patterns or recognisable features in random noise. What is needed is a noise function which will add clearly distinguishable global features while also adding the necessary amount of detail. Fractal functions have these abilities as can be seen in the well known Mandelbrot set, which has a distinct global shape that repeats, with small

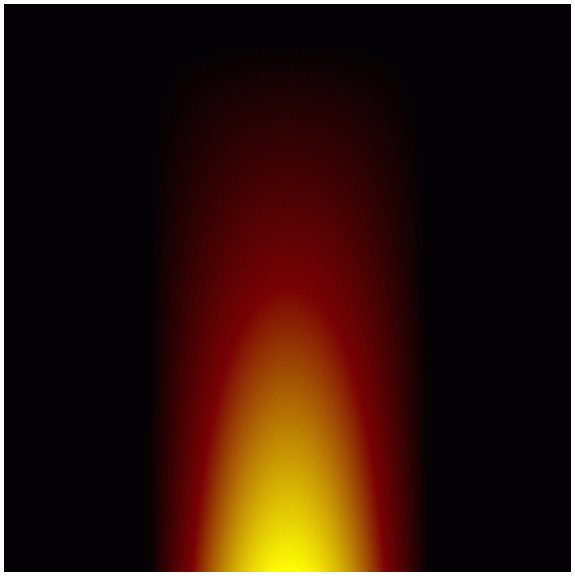


Figure 4.5: *The 'perfect' flame. Created by the simple formula (4.1).* †

changes, when zooming in on an area. Any amount of zooming is (theoretically) possible. A special kind of fractal functions are the fractal noise functions[1], where noise is used as an important part.

The turbulence function is such a fractal noise function. It uses an array of random values to create a turbulent result by using the following iterative formula:

$$turbulence(x) = \sum_{i=0}^{N} \frac{interpolated\_noise(2^i \lambda x)}{2^i} \tag{4.2}$$

where λ is a scaling factor to the pixel position x. The values $2^i \lambda x$ are used as indices in the noise array (modulo the number of noise samples the array contains), and the result is found by interpolating between the two nearest neighbours, which is done by the function *interpolated_noise*.

The interpolation of the array yields gives the desirable result, that input values lying 'close' (e.g. $|\lambda x_1 - \lambda x_2| < 0.1$) will result in turbulence values lying close. On the other hand, the summation assures that small scale detail appears. This makes the turbulence a fractal noise in the sense that zooming in on a smaller part of the output (sampling the x-values closer, or choosing a smaller λ) will reveal approximately the same appearance. The variable N controls the level of detail—the amount of zooming possible before the turbulent features disappear. The range of the output depends on the noise array. If this array contains random numbers between $-\frac{1}{2}$ and $\frac{1}{2}$ then: $\forall x : turbulence(x) \in [-1, 1]$.

The turbulence function has nothing to do with real-life turbulent wind fields. To simulate real turbulence you would need a detailed model of the scene and all wind fields in it. It does however produce good results as in figure 4.6, where the turbulence function has been used to disturb the 'perfect' flame from figure 4.5.

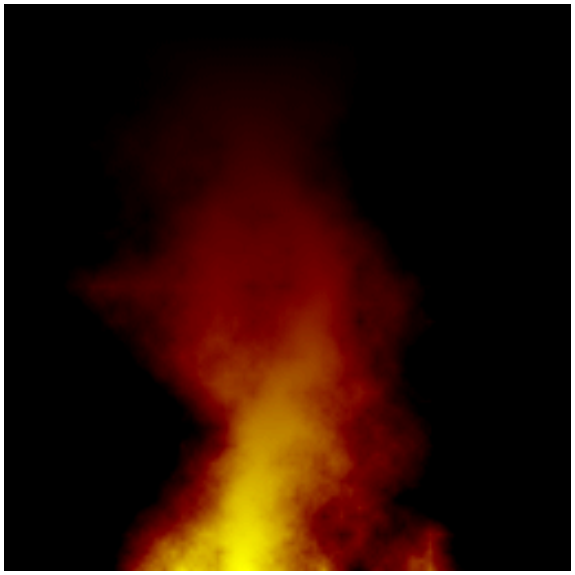


Figure 4.6: *The result of adding turbulence (4.3) to the 'perfect' flame from figure 4.5.* †

[1]For more information on fractal noise read [Mandelbrot and van Ness, 1968].

This has been done by extending the turbulence function to accept two dimensional input:

$$turbulence(x, y) = \sum_{i=0}^{N} \frac{interpolated\_noise(2^i \lambda x, 2^i \lambda y)}{2^i} \tag{4.3}$$

The noise array has been replaced by a two dimensional noise matrix, and therefore the function to interpolate noise has become two dimensional as well. The result of $turbulence(x, y)$ is then simply used as an offset to the x value in equation (4.1), that is:

$$output(x, y) = I(x + turbulence(x, y), y)$$

Animations can be easily obtained by extending the turbulence function further to take a third argument t—a time counter. The noise matrix must then become three dimensional, and so must the interpolation function. This algorithm is pretty fast (a couple of seconds per picture) but can *not* run real-time due to the high computational cost of the turbulence function. A little sequence from such an animation can be seen in figure 4.7.

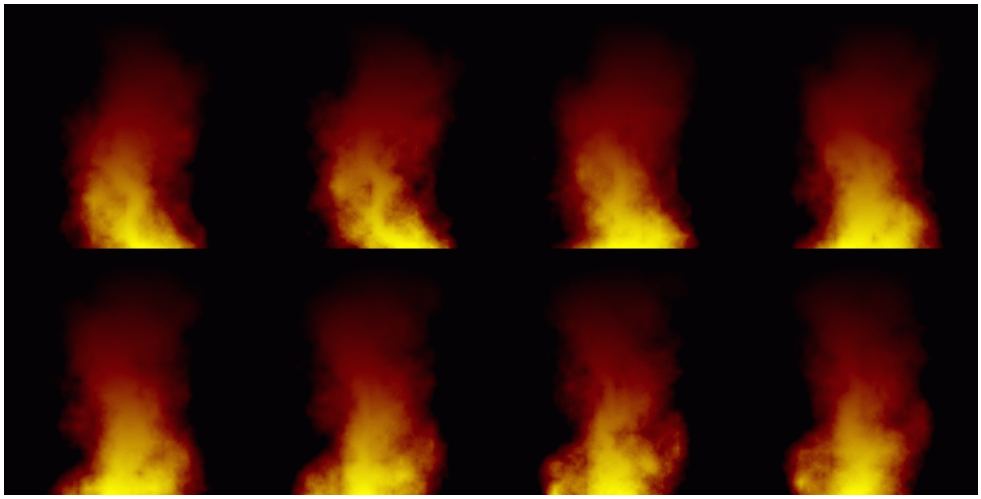


Figure 4.7: *Eight frames from an animation of the two dimensional turbulent fire.* †

To create a convincing fire animation using these turbulence calculations, it was necessary to employ a minor 'hack'. When looking at the animation from figure 4.7 you can spot twirls and distinct features moving upwards from frame to frame. This motion has been accomplished by iteratively subtracting a small amount from the y variable in equation 4.3 each frame, thus in fact moving the entire turbulence upwards. If this was not done, the distinct features would tend to remain at the same position, resulting in a warbling/boiling look.

Expansion to 3D

At this point an obvious thought would be to expand this 2D fire to 3D. The first step would then be to define the function that describes the 'perfect' flame in three dimensions. A very simple modification to equation 4.1 should do the trick:

$$I(x, y, z) = (1 - \frac{y}{yres}) \sin(\frac{r}{xres}\pi)$$

where $r = \sqrt{x^2 + z^2}$.

The formula for calculating turbulence must be equally expanded to take four arguments x, y, z, and t. Here it is important to note that the 'perfect' flame in 3D must be warped in both the x and the z parameter (where the 2D flame was only warped in the x parameter). This can be achieved by applying the turbulence formula twice using two different noise matrices.

If all this is done properly, we end up with a four dimensional function describing light intensities in any position $\mathbf{x}$, at any time t. The big problem is then how to visualise this. A simple solution would be to take each pixel in the output picture (so x and y are fixed), sample the intensity for a number of z-values and find the total intensity as a weighted sum of the samples. Although this is very simple it actually works, and is an example of what is commonly known as *'volume shading'*.

Volume shading plays a major role in our visualisation of fire and smoke, and the above described procedure for creating turbulent appearances is basically a simple version of the blob visualisation algorithm described in section 6.9.4.

4.2 2D Particle systems

As a completely different approach to the creation of digital fire, particle systems can be used. In this section we will give a small example of a two dimensional particle system, briefly describing what it consists of and how to use it. Particle systems are in general not restricted to two dimensions, and in fact the first known definition [Reeves, 1983] was stated in three dimensions. We have made a much more detailed description concerning three dimensional particle systems which can be found in section 5.2.

A particle system consists of two types of abstractions: An emitter and the particles which are emitted from it. These are controlled by a set of attributes, which could be: Position, size, transparency, lifetime, velocity, colour and shape. Figure 4.8 shows a very simple example with constant emitter velocity. Consider for instance a volcano (the emitter) spraying chunks of glowing lava (the particles) into the air, all of which fall down again. The lava chunks are "born" with a velocity vector that changes over time because of gravity. When they fall onto the ground the particles "die" and may create new particle systems consisting of smaller chunks of lava bouncing off, which then die and so on. This can be arbitrarily complicated.

Particle emitters are usually non-visualised abstract objects which can produce particles in any imaginable way, according to their shape. For instance particle emitters can be shaped as points, lines, rectangles, discs, and so on. Figure 4.8 is an example of a line emitter. Often the emitters are placed inside (or on) visible complex objects, as in the example with the volcano, that could be created by placing a disc shaped lava spewing particle emitter in the volcano opening.

The particles themselves can be visualised in many ways. For instance, lava particles could be dots, lines, spheres, or even bitmaps. Thus, by choosing a good visualisation for particles and moving them about in realistic ways the appearance of fire can be achieved.

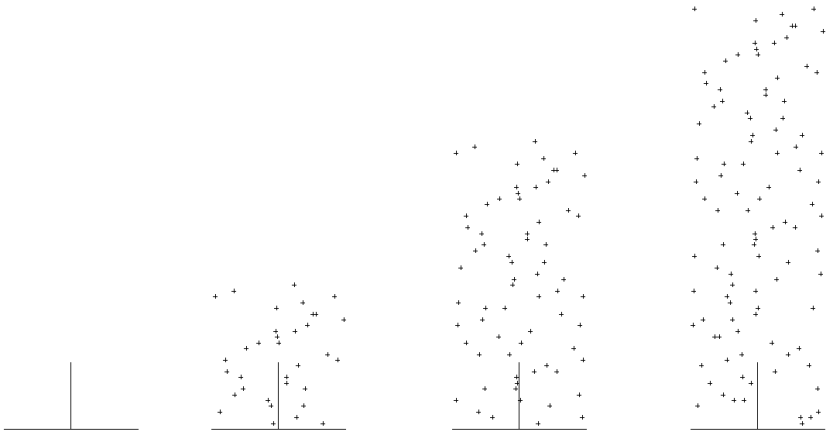


Figure 4.8: *Four frames from an animation of a two dimensional particle system at time $t = 0, 1, 2, 3$ seconds. The emitted particles in this example have constant velocity vectors and are not advected by gravity or other external forces for simplicity reasons. This can be seen by noticing the undisturbed pattern moving vertically which is created by the particles. The only change in the pattern occurs when particles "die" and are removed from the system. The particles have in this example a lifetime around $2\frac{1}{2}$ seconds. Notice the influence of this in the fourth frame.*

This can be seen as a simplification of the 3D "bitmap splatting" technique described later (in section 6.8.1).

What can be said specifically about the simple two dimensional model is that it is inherently without a third dimension, which eliminates perspective. This makes the overall rendering a bit faster. However, many kinds of three dimensional particle system rendering techniques do apply to 2D. Some techniques can and should be altered to obtain an increase in rendering speed since some of the rendering techniques are true three dimensional.

4.3 Summary

In this chapter we examined different ways of creating two dimensional fire visualisations. The methods were solely based on hacks, and have been refined by subjective decisions of "what looks good". The keyword here is '*advection*'. Both with the cellular automata fire (section 4.1) and with the turbulence shape warping (section 4.1.2) a very simple and dull standard fire was advected, creating a much more interesting result.

In the next two sections we will examine how to model, animate, and visualise fire in three dimensions. Even though these sections focus much more on achieving a 'correct' solution, advection of simple models still plays a major role.

Chapter 5

Modelling & animation of fire in 3D

"This video needs more
chicks and more fire!
Fire! Fire! <he he> Fire!"

Beavis & Butthead

Modelling of fire and smoke encompasses more than the physical three dimensional representation. It also involves the modelling of colours (texture), behaviour (such as spreading) and interaction with other objects, other phenomena, and light.

This section will start by describing ways of *simulating* (not *emulating*) some of the important features of fire and smoke. This is done by defining statistical representations of the desired phenomena. The representations have the advantage that they can add any amount of detail, while being defined by small amounts of input. In section 5.4 this is used to create a *turbulence matrix*, a set of data values useful for adding a turbulent appearance to an animation or visualisation. We have used this feature in our raytracer implementation—examples are available in section 8.

Furthermore we define particle systems in 5.2, and show how the gas motion in a room can be simulated (section 5.5) giving a very useful tool for animating particles. This motion deals with walls and other obstacles in the room, and has also been used in our raytracer.

5.1 Nondeterminism

When modelling natural phenomena people tend to use randomness to control the alterations of the current state. This has been the most widely used technique for "predicting" the phenomenon, since being introduced by [Founier et al., 1982]. Their approach, also known as *data amplification*, leans against the fact that instead of using a deterministic featuring function, it is much easier for the users (and the programmer), if they select a certain statistical profile for the phenomenon. So instead of having a function that returns a state of a phenomenon given only one parameter, namely time t, we then have a function that must have a state of a phenomenon and a Δt to create a new state for it. One of the large problems with nondeterminism is that it becomes very hard to compute in parallel, which is normally one of the nice features of computer graphics.

5.1.1 Stochastic modelling

Stochastic modelling has been used to model several natural phenomena. When modelling (semi transparent) fuzzy objects like clouds, fire, dust, mist, and smoke, a vast amount of memory and clock cycles are usually used without gaining considerable user control. To provide a better solution for this, let us look at stochastic modelling.

Jos Stam introduces a new way of modelling stochastically in his thesis [Stam, 1991]. His approach is to have models on two or more scales of visual detail: one macroscopic and one or more microscopical. This differs from the way natural phenomena are created when using fractal models, as these assume that the statistics are the same at all levels of the model. This is usually not true. If you look at a mountain from far away, you will only notice its large features, but when looking at a small portion of the mountain, it is obvious that it is made from different kinds of rock, and the vegetation is spread in certain

ways. When scrutinising at a special kind of moss, you will find a whole different set of features. So different kinds of statistics exist at different levels (in real life), why not use this in the computer model? Furthermore not everything should be defined stochastically, as this would give us no way of controlling the phenomenon, and thus no way of animating it. Therefore we need to be able to place constraints on the statistics at certain locations.

The model is introduced by [Stam, 1991], and later refined in [Stam, 1995], but we will focus on the first one, since it is relatively simple and demonstrates the basic idea clearly. The latter is more complicated and a discussion of it would be outside the scope of this thesis.

The *macroscopic level* is used for forming the general shape of the phenomenon and this is the part of the phenomenon that is user controlled. The user can model the form of the phenomenon by hand or by using a modelling technique that gives some modelling freedom but is controlled by physical means, for instance, by controlling a particle system (see section 5.2) that delivers this data to the model. The user has little influence on the result: He or she controls the largest features like origin of emission (if trying to model a gas jet), but never has to deal with physical means, such as interaction with solid objects. The microscopical levels are then applied to the model.

The *microscopical levels* are for finer details which also include colouring and translucency, all of which are too tedious for the user to specify. These small detail levels are called *second level statistics* (or in the case of turbulence: *second level turbulence*—see section 5.4) since they are usually modelled using statistical measures because many of the above mentioned phenomena have known statistics, such as correlation measures for their texture (see [Ebert et al., 1998], [Foley et al., 1990], [Stam, 1991], [Stam, 1995] or [Clausen, 1998]).

Stam's model combines these two (or more) modelling levels to render the phenomenon. The advantages of this model are better user controllability which makes it easier to animate, low memory usage, compatibility with the usual rendering techniques, and easy addition of small scale detail, which enhances its fire modelling capabilities.

5.1.2 The global scale—macroscopic level

The global scale of a phenomenon is what you see in a blurred picture of it. In other words: if all high frequencies are removed from the picture. You only see the crude, large features. In Stam's model, the user has the control of this part of the phenomenon, as he or she is allowed to specify its global appearance at certain locations (this can be modelled in many ways, for example by specific density functions or particle systems). This is done with n scalar values d_i at the corresponding locations $\mathbf{x}_i$ (see figure 5.1). Away from these locations, we would like the phenomenon to be *"well-behaved"*. In other words, we have to choose a function that interpolates smoothly between these constraints according to their location. Such a function, $S(\mathbf{x})$ must behave at least so that $S(\mathbf{x}_i) = d_i$ for $i = 1, \ldots, n$. The interpolation method must be selected with caution. For instance a Lagrange interpolation is unsuitable here as it may produce unwanted results (see [Kincaid and Cheney, 1991] for an explanation of this method and its drawbacks). *Surface spline* interpolation has been used by Bookstein, Meinguet and Szeliski & Terzopoulos with acceptable results (see [Stam, 1991]). A better way of doing this must be found if the phenomenon is more complex than the flame from a candle. This is especially a problem when the number of locations starts to increase considerably as it will do in

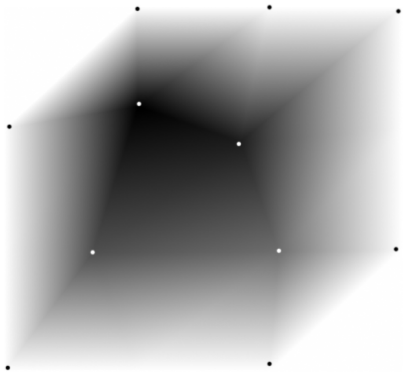


Figure 5.1: *An example of a "global shape". The object is made from scalar values. The internal values are found by interpolation or* kriging.

almost any case with fire. [Stam, 1991] introduces a new approach to this as he looks at the interpolation as an *estimation* problem solved by *kriging*, which is a special way of interpolating the scalar values. When this method is used a drastically reduction of spent clock cycles is achieved. However, a thorough treatment of this is outside the scope of this thesis.

5.1.3 Small scale detail—microscopical level

Small scale detail is added to give the phenomenon its complex or rough features (the high frequency). The easiest way to understand this is to think of it as the phenomenon's texture. There are several ways to create this; [Lewis, 1989] and [Stam, 1991] present some examples, some of which are briefly discussed here. The general idea by doing just so is to give a short introduction to the matter and to show that the small scale of detail can be of arbitrary complexity.

Random Functions

A random field $R(\mathbf{x})$ can be used for second level statistics and as seen in the works of [Lewis, 1989], a *model-directed synthesis*[1] function $W(\mathbf{x})$ approximates this well. If $W(\mathbf{x})$ is expanded to include a time parameter, it should be independent of earlier calculated values of W, so that it is possible to find the noise at any time. This is not always how random functions behave, but it is essential if we want to pursue parallel rendering.

Random functions are mostly functions that are chosen or found *ad hoc*, many of them having no resemblance with either physics or real world observations, but are merely experimental hacks that feature fast evaluation and fine results.

Spectral Sums

[Yaglom, 1986] has proven that random fields can be arbitrarily approximated by sums of simple random functions. The second level statistics can therefore also be described by a

[1] In which sampling and construction of noise occur only at points where the noise value is required, rather than over a regular sampled region of space. This is an attempt to minimise computational efforts.

spectral density function. The random function can be given by:

$$W(\mathbf{x}) = \sum_{i=N_{min}}^{N_{max}} A_i W_i(\mathbf{x}), A_i \in \mathbb{R}$$

This is a sum of band-limited random functions W_i, all of which have a narrow spectrum and are (ideally) mutually disjointed so that all of the $A_i W_i(\mathbf{x})$ together model the entire spectrum. The error of the approximation is determined by the bounds of the summation. It may not be desirable to demand a very small error due to its enormous computational cost as it increases with larger bounds (in the same way that it is seen with Taylor expansion). Thus, it yields yet another choice between rendering speed and image quality.

Perlin's Noise

[Perlin, 1985] suggested a class of random functions now widely used in computer graphics. An approximation of white noise $N(\mathbf{x})$ is used to create more complex functions. Perlin's noise function is a sum of scaled samplings of $N(\mathbf{x})$, as:

$$W(\mathbf{x}) = \sum_{i=N_{min}}^{N_{max}} \frac{N(2^i \mathbf{x})}{2^i}.$$

This is basically the same as equation 4.2 that was used to create the two dimensional turbulence in section 4.1. However N_{min} and N_{max} must be selected with caution. [Stam, 1991] suggests that if the resolution of the image is r, then $N_{min} = \log(1/r)$ and $N_{max} = \log(r)+1$ should be safe choices.

Gardner's Texture

[Gardner, 1985] has developed a three-dimensional texture function to model clouds. The function is generally an *ad hoc* model and does not originate from real-clouds physics. Basically Gardner mapped this texture function onto single or sets of ellipsoids and got really nice results, most possible due to the fact that the model does include translucency in which it differed from earlier works. The texturing function is written as:

$$W(x, y, z) = k \sum_{i=1}^{n} F_i(x, y, z) \sum_{i=1}^{n} F_i(y, x, z)$$

where

$$F_i(u, v, w) = C_i \sin(\omega - iu + \frac{\pi \sin(\omega_{i-1} v)}{2} + \pi \sin(\frac{\omega_i w}{2})).$$

The coefficients are "characteristic" values (*ad hoc* values), Gardner has achieved good results for clouds with

$$C_i = (\frac{1}{\sqrt{2}})^i C_0, \quad \omega_i = 2^i \omega_0.$$

For further discussion of the equation, C_0, and ω_0, please refer to [Gardner, 1985]. This model for second-level statistics has presently to our knowledge only been used to model clouds, which is applicable for smoke. Unfortunately we have not had the time to pursue this to find characteristic values for C_0 and ω_0 to model fire, if there are any. This we must add to the list of future projects.

Other second level statistics

The above second level statistics are just examples. Many others have been found. Jos Stam furthermore mentions *the Weierstrass-Mandelbrot function*, which is a generalisation of one of Weierstrass' functions resulting in a *fractional Brownian motion* which is created by a superposition of sinusoids, *superposition of one-dimensional functions*, which is an even more generalised version of the Weierstras-Mandelbrot function. *Sparse convolution* found in [Lewis, 1989] deals with *Poisson noise processes* which have the applicability of lookup tables for the convolution kernels to gain speed ups.

Both [Kajiya, 1989] and [Perlin and Hoffert, 1989] have used *thick textures*. You may know the digital teddy bear made by Kajiya with its most impressive fur (if not, it can be found in one of the colour plates in [Foley et al., 1990]). But Kajiya can model things other than fur, such as gaseous phenomena as long as the user takes care of texels[2] that do not blend particularly well. The method is unfortunately, computationally expensive. [Perlin and Hoffert, 1989] used what they called "hypertexture" to model fire balls (as well as fur and glass), but the drawback of this method is its expensive volume-rendering, since their technique depends on density fields only.

5.2 Particles

With particle systems it becomes possible to define the behaviour of a phenomenon and thereby reaching a visualisation, without actually knowing in advance how this will look. The user should be able to define the system in a way that gives control of the overall behaviour, while the small scale details are left to be handled by the model. As an example, when trying to model a column of smoke rising from a chimney, the user should be able to define the colour and thickness of the smoke, in which direction the wind blows, and the amount of billowing (and perhaps a few more variables). Exactly *where* this billowing occurs and how it is done is irrelevant, as long as it looks good to the user (the stochastic modelling described in section 5.1 with the application of second level statistics is an example).

5.2.1 "What is a particle system?"

Consider snow falling in big snowflakes on a relatively warm December day with temperatures just above freezing. This is a good example of a particle system from real life. Each snowflake (or snow*particle*) has its own position and movement. Furthermore it is controlled by a set of (physical) rules. The gravity pulls it towards the ground, wind affects its motion as well, and if two flakes collide they might stick together, or they might break into smaller flakes. A snow *particle system* would consist of a lot of snow particles, in this case defined solely by their positions in space, two rules for movement (gravity and wind), and a collision rule.

This example is obviously very simplified, and usually the representation of a particle consists of a lot more information than just its position. For instance colour, temperature, size, shape, etc. A particle can even be a particle system in it self. In the snowfall example each snowflake could be a particle system, consisting of frozen water particles.

[2] *Texels* are cubes with certain functions that describe small parts of the surface (*microsurfaces*) for the object.

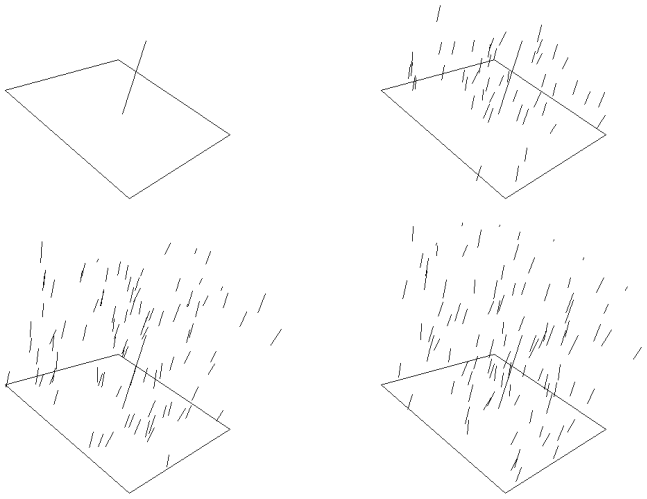


Figure 5.2: *Four frames from an animation of a three dimensional particle system. The emitted particles in this example have velocity vectors that are advected by an internal force which makes the particles spread and move away from the emitter. This gives a chaotic impression rather than that of figure 4.8 which is unadvected. The particles can also be advected by external forces like gravity, wind and convection. This has not been applied here in order to keep the example simple.*

A very common particle attribute is the particle's lifetime. The lifetime defines how many frames or seconds the particle will live, and can be used to remove particles that are no longer necessary in the animation. It can be a simple counter which is incremented in each frame, or it could depend on external event - such as the particle leaving the screen with no hope of returning. A snowflake particle would die if it was warmed up and melted. This implies that the lifetime is controlled by the amount of heat the particle receives. This is by no means a trivial incrementation, as the temperature is much higher on the ground than in the air.

When all the initial particles have been killed the animation soon becomes dull unless new particles are born. Normally this is done by applying some sort of statistical process to the number of new births and their initial values. A very simple but often used model is to keep the number of new births per frame constant, and let the new particles come to life somewhere randomly inside a defined area—the *particle emitter*. Another simple model is to keep a constant number of particles, i.e. each time a particle dies it is reborn. If the density of particles should be bigger in some points than in others, a Gaussian distribution is often used to decide the particle's place of birth.

5.2.2 Reeves' definition

In 1983 William T. Reeves [Reeves, 1983] from Lucasfilm Ltd. was charged with the job of creating a computer rendered sequence to the film *Star Trek II: The Wrath of Khan*, please look at figure 1.3 on page 6. The sequence should show a computer simulation of a bomb (called the Genesis bomb) impacting on a dead planet, creating a wall of fire and

explosions spreading across it, and thereby creating life on the planet[3]. Since it should look like a computer simulation, the result did not have to be too realistic looking - actually the opposite would be preferred[4], but needless to say it should be impressive. He solved the problem by creating a program, (using all the hacks of the trade), where huge amounts of tiny objects flew around in a three dimensional world, each adding a tiny amount of reddish light to the calculated colour in the two dimensional pixel they covered. He called these objects 'particles', and gave them the following attributes.

Position	Velocity (both speed and direction)
Size	Colour
Transparency	Shape
Lifetime	

The number of new particles for each frame, and their initial values were controlled by adding small random variations to a well defined mean position, mean velocity, mean number of new births etc. In an effort to optimise the rendering process, the birthrate of the particles depended on the actual screen area covered by the particle system. In that way he could avoid spending unnecessary time rendering particles that could not add any further detail. He then went on to define a particle hierarchy as a particle system where the particles themselves were particle systems.

The spread of fire across the planets surface was simulated by a particle system located at the impact point of the Genesis Bomb. New particles were born in expanding concentric rings on the surface, giving the illusion of a moving firewall. Each of these particles were themselves particle systems, sending new particles flying away from the planet, thus giving the illusion of explosions. The particles did not emit any light, so to create the effect of the fire lighting up the surroundings, a big expanding spotlight was used. To achieve convincing results about 750.000 particles were used [Reeves, 1983].

5.3 Fields

To avoid the tedious simulation of an entire world of objects with its interacting atoms and molecules constrained by the laws of physics, the world can be simulated at a larger scale. If not, the limits of computer memory and computational capacity assure that only small worlds (small scenes) can be simulated. In other words is it clear that a simulation of a large world cannot be done without loss of detail.

A choice of scale must be somewhere between molecular scale and object scale (where objects are treated as the smallest entities). Think of objects subdivided into finite sets of sub-objects that ensure a certain level of detail or quality. An easy choice is to break every object down to a set of simple three dimensional graphics primitives (such as triangles and spheres) and use only these in a simulation of the world.

This approach only takes care of solid objects. The rest of the world is now void and without reference (no pun intended). Some problems arise because of this, all of which are related to transportation of energy through air. For instance secondary spread of fire is impossible to simulate due to the lack of temperature transportation information. Drag and turbulence caused by moving objects is also unavailable. Flames and smoke are

[3]Hollywood bombs never behave in realistic ways.

[4]Hollywood computers never behave in realistic ways.

now only possible to remake as hacks—an example could be a procedurally defined and animated particle system. [Stam, 1991] uses particle system modifiers to simulate wind and turbulence. An example of this could be the turbulence caused by an exhaust pipe.

To solve this situation the same subdivision can be applied to the entire three dimensional scene, thus dividing the world representation up in sub-areas. These subdivisions are in the computer graphics literature referred to as *fields*[5]. A field has, just like a particle system, certain information concerning the scene in each cube (or *voxel*) representing an area of the scene. This information facilitates simulation (at larger scale) of the mentioned problems. What these are precisely and how they relate to the simulation of fire will be discussed in the following section.

Fields could, if not used with caution, introduce yet another vast usage of the computers memory. The grid (also called the *voxel environment*), should both surround the entire scene and its cubicles should be small enough to avoid immediate spread of fire through solid objects (e.g. a wall or a steel plate), and should facilitate smoke and flames travelling through small holes. Consider a three dimensional grid surrounding the parts of a scene that might end up burning or affected by the fire (most likely the entire scene). If a grid does not surround the entire scene, its boundaries should at least be where the fire is *incapable* of spreading to. An example of this could be your standard snow cabin with fireplace, mantlepiece, poker and the like. The fire is *not* needed to be simulated outside the fireplace, unless you are visualising the works of an arsonist.

There are several types of fields. We will discuss the following representation types:

Temperature fields This type of field is used to control spread of temperature, which has a great influence on the spread of fire. A temperature field will support interaction between smoke and fire from multiple sources. The lack of these features are the major disadvantages of the models introduced by [Watt and Watt, 1992] and [Stam, 1995]. They also allow interaction with other types of fields, for instance pressure and motion fields (see below). The spread of temperature is not simulated easily though, but it can be dealt with as we shall see in section 5.5.1. Figure 5.3 shows a visualisation of a temperature field.

Motion fields Motion fields are mainly used to simulate or control the motion of particles in a scene, but will also support calculations of interaction between gasses, or even effects like gravity or magnetism. (The latter is of no interest to this thesis, though). The spread and mixing of hot turbulent gasses (such as flames) are modelled using a combination of motion fields and temperature fields in section 5.5.1.

Pressure fields When substance or solid material is burning violently or small explosions are caused by sudden released fuel, local pressure is build up and released causing disturbance in the surroundings including temperature and motion fields. This can be simulated using a pressure field (see figure 5.4). Pressure fields can be used when calculating *volume-of-air* ratios to decide if a solution is too rich or lean (see figure 3.1).

Fuel fields We think of this kind of field as one inside an object. They can also be applied to the entire 3D scene if a visualisation of a lit gas is desired. The features this type

[5] In the scope of physics the term 'field' is normally used for a continuous function describing properties within in an volumetric area. This is why a *density field* is not a discrete subdivision of a scene (see section 6.9.1).

of field holds, however, are applicable on both objects and "non-matter objects". An object that burns will most probably not do so evenly, since it does not contain fuel homogeneously (a log will have different concentrations of wood oils in its body—another object might be inflammable in some parts). In order to simulate this we propose the usage of a *fuel map*, which for objects can be used in 2D (aligned with the objects surface texture maps) and 3D (a grid surrounding the object or the scene). The fuel map is thought of as an attribute to the object or material. Initially the fuel map is setup with fuel, and when the objects fuel is ignited the fire is allowed to spread while it is fed from the fuel map (see figure 5.5).

Fuel maps work very well with temperature fields and are easy to comprehend and setup by a user. This can be done easily in 2D, but it is slightly difficult to do so in three dimensions—though a 3D painting program should not be extremely difficult to implement or handle. Other effects are easy to add, for instance when the fire burns, the polygons' texture map should be coloured appropriately as they would be by soot.

Fuel maps can be animated bitmaps (prerendered or calculated in parallel), which can create some stunning results.

Surface degradation At a certain time fire burning from a surface made of a flammable material will start to receive flammable vapour from the interior of the object which leads to a change in the structure of the surface. Imagine for a moment (or get inspiration from taking a glance at figure 5.6 on page 54) a wooden table with a burning liquid on it in the shape of a logo. When the fire dies out there should then be a burned logo on the table surface. This effect can be simulated quite simply by using of displacement mapping (described briefly in [Foley et al., 1990]).

The displacement is easily derived from the fuel map and temperature map. These changes are easily stored onto the fuel map for each frame, and a texture map is also easy to derive from these maps.

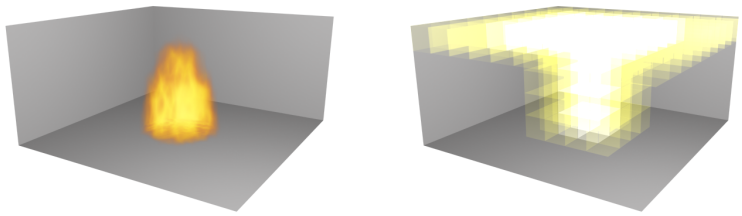


Figure 5.3: *An example of a temperature field. The voxel environment size is* $10 \times 10 \times 5$.

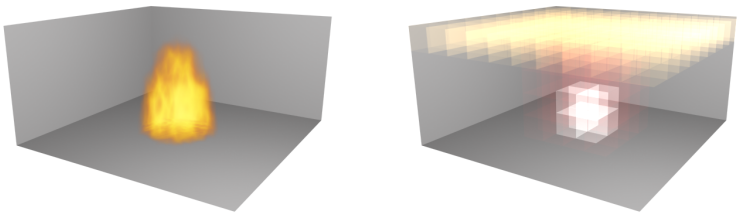


Figure 5.4: *An example of a pressure field. The voxel environment size is* $10 \times 10 \times 5$.

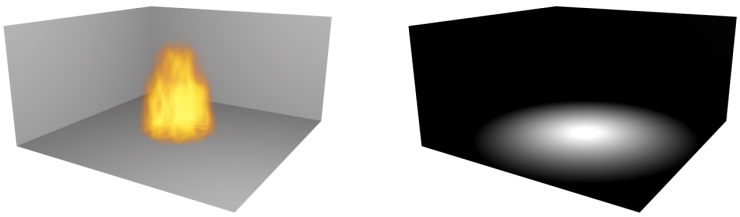


Figure 5.5: *An example of a fuel map. The scene with a burning fire and the fuel map—mapped onto the surface. Resolution of the fuel map is* 300×300 *pixels.*

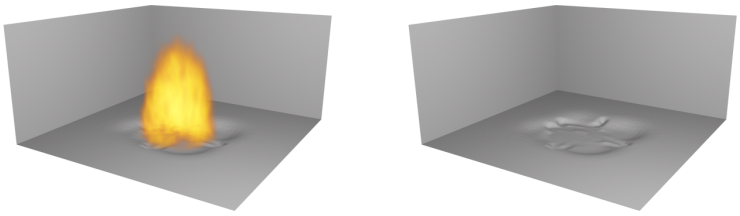


Figure 5.6: *An example of a displacement map. This is the same scene from figure 5.5 and what is left of the surface. The resolution of the map is* 100×100 *pixels.*

5.4 Turbulence

In section 4.1 (equation 4.3) a turbulence function was created to give life to a simple image of a flame. This function took a pixel coordinate (x, y) as input, and returned a scalar value. To create the appearance of a changing turbulent wind field, four dimensional input is needed (a spatial and a temporal component) and three dimensional output (a vector describing the direction and size of the wind force).

A very simple solution is to take three turbulence functions from section 4.1 and simply expand them to four dimensions. Although this is a useful approach, it is also slow because the turbulence calculations are computationally expensive. As the turbulence plays a

major role in the visual characteristics and animation of fire or smoke, this could have a big impact on the rendering times in an implementation, especially if turbulence is desired to create special effects in real time applications (such as computer games). To calculate the turbulence each time it is needed would simply be impossible. Here it would be much better with a solution where the turbulence can be found in real time, perhaps even while the user modifies the parameters. Such a very fast solution can be obtained by sampling the turbulence function at specific points in space and time, and storing the results in a four dimensional array. The turbulence can then be computed at any point by interpolating the values in the array, at the cost of detail.

This turbulence array has two major drawbacks, though. First of all it is extremely memory consuming. A simple $64 \times 64 \times 64 \times 64$ array containing three dimensional vectors of floating points will take up 192 MByte of memory (if floating point numbers take up 4 bytes—not 8). Obviously the size of the array must be greatly reduced to be of any use. Luckily even small arrays of dimension $16 \times 16 \times 16 \times 16$ (768 K bytes) should be able to yield good results, as the turbulence itself is not actually seen—only its effect on the surrounding particles. Such a small turbulence matrix has been used to create the scenes both in [Stam, 1995] and in chapter 8. When looking at the figures it is hard (if not impossible) to see, that that turbulence array used is so small.

Figure 5.7 shows the first frames from an animation of particles being blown around in such a turbulent wind field. It is clear, that although the turbulence array is small, convincing results are obtained.

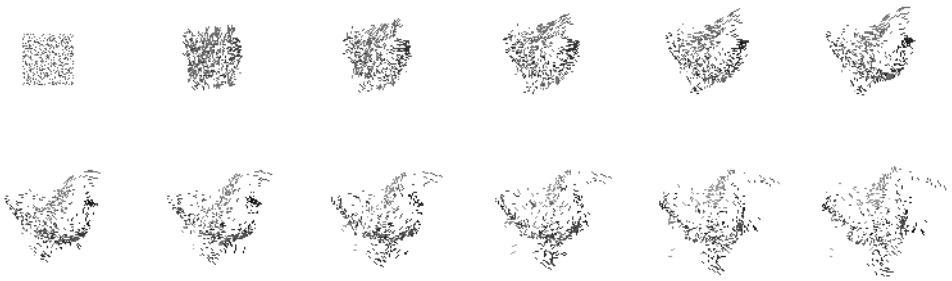


Figure 5.7: *Particles being animated by turbulence.*

The second major drawback of the above described turbulence array is closely connected with the fact, that the array must be kept so small: To create the appearance of turbulence everywhere in space and time, the world coordinates should be taken modulo 16 and then interpolated in the array. But as the turbulence function does not create periodic turbulence, this method suffers from noticeable 'jumps' if used in animations. Also, the formula for calculating turbulence—equation (4.3)—seems to favour turbulence on the line $x = y = z = t$.

To overcome these difficulties [Stam and Fiume, 1993] developed an algorithm for creating periodic turbulence arrays. In the following section the results are given. For details of the proof, see [Stam and Fiume, 1993] and [Stam, 1995].

5.4.1 Creating the turbulence array

The algorithm uses results from Fourier analysis, so let us start with a short recapitulation. More detail can be found in the many introductory books on Fourier transforms, for instance [Brigham, 1988].

A brief informal description of Fourier Analyses

For the function $g(t)$ the *Fourier transform* $\hat{g}(\omega)$ is defined as:

$$\hat{g}(\omega) = \int_{\mathbb{R}} g(t) e^{-i2\pi\omega t}\, dt \quad \hat{g} : \mathbb{C} \curvearrowright \mathbb{C}$$

Using the same terminology, the *inverse Fourier transform* is defined as:

$$g(t) = \int_{\mathbb{R}} \hat{g}(\omega) e^{i2\pi\omega t}\, d\omega \quad g : \mathbb{C} \curvearrowright \mathbb{C}$$

The Fourier Transform can be interpreted as a transformation from the time domain to the frequency domain, that is if $g(t)$ is a function of time (t is a time parameter), then $\hat{g}(\omega)$ shows the distribution of frequencies in $g(t)$, and ω is a frequency parameter. This interpretation has proven very useful in sound analysis to recognise specific frequencies in music. If the Fourier transform is applied to an interval $[a, b] \subset \mathbb{R}$, the result becomes an approximation that is periodic in $\mathbb{C}$ with length $|b - a|$, that is: $\hat{g}(\omega + |b - a|) = \hat{g}(\omega)$.

In many applications it is often practical to *filter* a function. This is done by *convolving* a filter function H with the input function g, using the following definition of convolution:

$$f(t) = \int_{\mathbb{R}} H(t - y) g(y)\, dy$$

The following useful equation is derived using the Fourier transform:

$$\hat{f}(\omega) = \hat{H}(\omega)\hat{g}(\omega) \tag{5.1}$$

The above equations are given in the one dimensional continuous case. The theory is easily modified to the multidimensional and discrete cases. The Fourier Transform of an n-dimensional function can be found by applying n one dimensional Fourier transforms successively [Brigham, 1988]. Likewise for the inverse Fourier transform. To calculate both the Fourier transform or the inverse Fourier transform for discrete input, an algorithm called the Fast Fourier Transform is normally used. This algorithm is explained in [Brigham, 1988] and C-implementations for one and two dimensions are presented in [Press et al., 1992].

The Fourier transform of a real function $f \curvearrowright \mathbb{R}$, is a complex function $\hat{f} \curvearrowright \mathbb{C}$. A number of properties hold for f and $\hat{f}$. The algorithm in the next section uses the property that if $f(t)$ is a real valued function, then the real part of $\hat{f}(\omega)$ is even, and the imaginary part is odd[6]. This is also true for the inverse Fourier transform, thus providing a way to ensure that the result of an inverse Fourier transform will be a real valued function.

[6] A function f is even if $f(-t) = f(t)$. f is odd if $f(-t) = -f(t)$

Creating turbulence using Fourier transforms

As previously mentioned, a turbulence array was created in section 4.1 by using an array of random values as input to a turbulence function. This can be compared to a discrete convolution of the noise array. In general, noise of any appearance can be created by convolving an array of white noise with a filter [Stam and Fiume, 1993] (the problem is 'only' to find a filter that creates the needed noise). If this convolution is done as in equation (5.1) an inverse Fourier transform of the resulting function $\hat{f}(\omega)$ will result in a periodic function $f(t)$. If $\hat{f}(\omega)$ is modified in such a way that the real part is even and the imaginary part is odd, then $f(t)$ becomes a real valued periodic function.

So we need to create $\hat{g}(\omega)$, which should be the Fourier transform of random noise, and also we need to find $\hat{H}(\omega)$, the Fourier transform of a filter, which is useful for transforming white noise to turbulent noise (when applied as in equation (5.1)).

Let us start by finding $\hat{g}(\omega)$. This could be calculated by actually creating a noise array and applying the Fourier transfom, but a *spectral representation*[7] of noise can actually be found directly with a little thought. Intuitively white noise consists of a lot of random frequencies of random amplitude. As an approximation you could say that each possible frequency is represented by a random complex number: $\hat{g}(\omega) = a \cdot e^{i2\pi\psi}$, where a is a random amplitude (for instance following a Gaussian distribution), and ψ is a random phase.

It is important to remember that the turbulence array we want to create should be four dimensional. This is necessary because we need to find the turbulence in any position in space (3 dimensions) at any time (one extra dimension). That is: $\hat{g}$ must be four dimensional $\hat{g}(\omega_1, \omega_2, \omega_3, \omega_4)$. After creating this four dimensional array of random frequencies, the creation of a convincing turbulence only depends on the filter $\hat{\mathbf{H}}$. This filter is now written in bold face, as it should return a three dimensional vector—the direction (and length) of the turbulence in the given position. [Stam and Fiume, 1993] show, that a good filter in the spectral domain can be written as

$$\hat{\mathbf{H}}(\mathbf{s}, t) = \Phi(s, t)\mathbf{P}(\mathbf{s})$$

where

$\mathbf{s}$	is	a vector $(s_1, s_2, s_3)^T$
s	is	$\|\|\mathbf{s}\|\|$
t	is	a time variable
Φ	is	a function for the turbulence's energy spectrum

$\mathbf{P}$ is defined as

$$\mathbf{P}(\mathbf{s}) = (\mathbf{I} - \frac{\mathbf{s}\mathbf{s}^T}{s^2})$$

where $\mathbf{I}$ is the 3×3 unity matrix.

[7] A spectral representation of a function is a description of the frequencies represented in the function.

The only undefined function is $\Phi(s,t)$, which have to be chosen in a way that describes the needed features of the turbulence. $\Phi(s,t)$ is in fact the *energy spectrum* for the resulting turbulence, which means that Φ describes the overall behaviour of the turbulence in the spectral domain by deciding which frequencies (s) have the most energy, and how this energy changes in time (t).

Intuitively turbulence is mainly a low frequency noise, so the energy spectrum should be high for small values of s and decrease for higher values. Also the small scale high frequency noise should be less correlated in time (small scale fluctuations change faster than the global appearances), so the energy spectrum should also decrease as t increases.

An intuitive guess at an energy spectrum could then be:

$$\Phi(s,t) = \frac{C}{1 + s^N + t^M}$$

where C, N and M are user defined constants.

In [Stam and Fiume, 1993] statistical fluid mechanics is used to come up with the following solution:

$$\Phi(s,t) = \sqrt{\frac{C_2 \exp(\frac{-t^2}{2s^2})}{s^{14/3}}}, \text{ for } s > S_0$$

where C_2 and S_0 are user defined constants. Stam suggests a value of C_2 around $\frac{1.5}{4\pi\sqrt{2\pi}}$, which seems to work fine. For values of s smaller than S_0, $\Phi(s,t)$ is set to zero. This way S_0 defines the 'slowest' movement of the turbulence.

The algorithm

Using the above theory, we got the following algorithm for creating a turbulence array $\mathbf{f}$. It is assumed that the spatial resolution is M and the temporal resolution is M_t. The array $\hat{\mathbf{f}} = (\hat{f}_1, \hat{f}_2, \hat{f}_3)^T$ is created so that the real part is even and the imaginary part is odd. This is done using the complex conjugate (*). In this way the inverse Fourier transform of $\hat{\mathbf{f}}$ will be real valued and periodic.

```
for i,j,k ∈ {0,...,M/2 − 1}
    for l ∈ {0,...,M_t/2}
        Compute f̂_{i,j,k,l}
        f̂_{M−1−i,M−1−j,M−1−k,M_t−1−l} = f̂*_{i,j,k,l}    // creates the correct symmetry
for i,j,k ∈ {0, M/2}
    Set the imaginary part of f̂_{i,j,k,0} and f̂_{i,j,k,M_t/2} to zero
```

The algorithm to compute $\hat{\mathbf{f}}_{i,j,k,l}$ is as follows:

generate three Gaussian random numbers A_1, A_2 and A_3.
generate three random numbers ψ_1, ψ_2 and ψ_3.
$\hat{g}_1 = A_1 e^{i2\pi\psi_1}$, $\hat{g}_2 = A_2 e^{i2\pi\psi_2}$ and $\hat{g}_3 = A_3 e^{i2\pi\psi_3}$
$s_1 = 2i/M$, $s_2 = 2j/M$, $s_3 = 2k/M$ and $t = 2l/M_t$
$s = \sqrt{s_1^2 + s_2^2 + s_3^2}$
$(\hat{f}_{i,j,k,l})_1 = \Phi(s,t)((1 - \frac{s_1^2}{s^2})\hat{g}_1 - \frac{s_1 s_2}{s^2}\hat{g}_2 - \frac{s_1 s_3}{s^2}\hat{g}_3)$
$(\hat{f}_{i,j,k,l})_2 = \Phi(s,t)(-\frac{s_2 s_1}{s^2}\hat{g}_1 + (1 - \frac{s_2^2}{s^2})\hat{g}_2 - \frac{s_2 s_3}{s^2}\hat{g}_3)$
$(\hat{f}_{i,j,k,l})_3 = \Phi(s,t)(-\frac{s_3 s_1}{s^2}\hat{g}_1 - \frac{s_3 s_2}{s^2}\hat{g}_2 + (1 - \frac{s_3^2}{s^2})\hat{g}_3)$

With the term 'a *Gaussian random number*' is meant an instance of a random variable with a Gaussian distribution. Such a number can be approximated by adding N random numbers from the interval $[0, 1]$. If $N/2$ is then subtracted the mean will be zero. The number can then be scaled to fit in any desired range.

After completing the above algorithm, the turbulence array is obtained by performing an inverse FFT on $\hat{f}_1$, $\hat{f}_2$, and $\hat{f}_3$.

5.5 Animation of particle systems

The turbulence array just created is very useful for moving particles in a turbulent looking manner. This model is, however, inadequate in at least two ways: first of all the relatively small turbulence array causes a lack of detail, which will become evident when the observer is relatively close. Second, the turbulence just stays in place, perhaps rolling a bit back and forth like water in a bucket being shaken, but not drifting about like smoke or rising like fire.

The first problem can be solved by adding a second level of small scale turbulence [Stam and Fiume, 1993]. In this way the big general motion is controlled by the first turbulence array, while the second array add detail without affecting the general appearance. This is consistent with the way we normally conceive turbulent phenomena. For instance a cloud could be described as 'a fluffy thing', but as we move closer it becomes apparent that even the 'fluffiness' is fluffy. In this way the turbulence is comparable with fractal noise, as the different levels of turbulence gives a local repetition of the global features (section 5.1).

The second problem is the problem of making the global motion of the particles appear realistic. In the case of fire and smoke, this motion should be decided by external forces such as wind or walls or air pressure, and by internal forces such as convection caused by the temperature. In other words the global motion should model the way the turbulent gas interacts with the surroundings in a scene. Because of the possibility of adding second level turbulence, only needs to be modelled on a large scale.

A very simple solution could be to add an upwards movement to all particles, thus simulating the effect of hot air rising. The speed could be constant or depend on the temperature of each particle. If the temperature is involved, heat transfer theory can be used to let particles receive and radiate heat on the surroundings and other particles. In this way the particles themselves play a major part in how heat is propagated in the scene. Unfortunately this solution is too simple to facilitate interaction between the gas and the scene. [Stam, 1991] suggests a design in which the animator is given full control of the global motion. In an interactive 3D environment the animator draws a global motion field

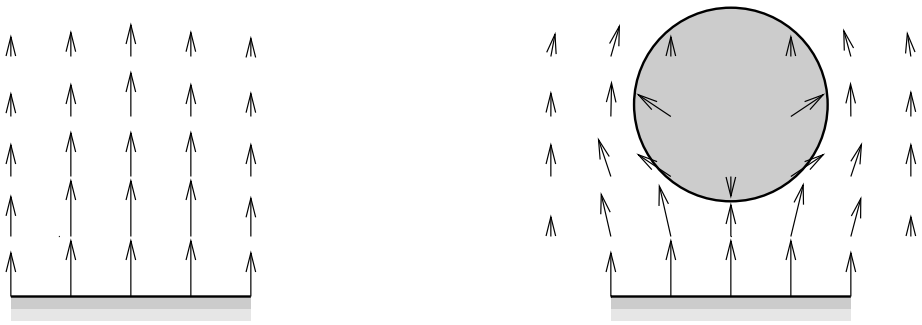


Figure 5.8: *Two example motion fields, as they could be drawn by the animator. The one on the left models a simple fire burning from a surface. In the example on the right a sphere is placed in the middle of the flame. Notice the way the motion field must be constructed to 'push' the particles away from the sphere, forcing them to skim along the surface.*

directly in the scene, as is shown in figure 5.8 (simplified to 2D). This is then used to interpolate the global movement of the particles, to which the turbulence is added.

The advantage of this method is, that it leaves the animator with full control of the motion, enabling 'weird' special effects such as bouncing fire or smoke drifting in circles. Furthermore the motion field and turbulence interpolations are so fast, that real time previews of the resulting particle animation are feasible. The downside is that great skill and knowledge is needed by the animator to make the motion appear realistic in complex scenes. And even in a simple scene it is tedious to be required to draw a complete motion field (consider the scene in figure 1.1 on page 5).

Another approach could be to use the laws of physics to calculate an approximation to the global motion, thereby taking away the control from the animator. A solution to this approach would be especially useful whenever realism is a main priority. In theory it could in fact make the turbulence calculations of section 5.4 superfluous, as a perfect solution would naturally incorporate the turbulence in a scene.

Recently [Foster and Metaxas, 1997b] proposed a solution for calculating the global motion of a hot turbulent gas. In their method the particles are just used as a visualisation tool for the gas. The heat propagation is handled entirely by the algorithm, which also takes interaction between the gas and solid objects into account. The algorithm has been included in our raytracer implementation, enabling automatic calculations of realistic motion fields, as the one used in figure 5.9 to move the dark sphere-shaped particles. In the following section this algorithm will be described along with the physics involved.

5.5.1 Modelling the motion of a hot, turbulent gas

In [Foster and Metaxas, 1997b] a distinction is made between solid objects and the surrounding air, but no distinction is made between hot turbulent gasses and 'passive' cold air. A hot turbulent gas (such as a flame) is simply a volume of air with a higher temperature and thus probably a greater motion.

Consider an old fashioned steam engine venting a jet of hot gas from its boiler. As it mixes with the surrounding air the surface of the gas will be slowed and cooled down, *dragging* behind and thereby creating the well known rotational turbulent motion. (At

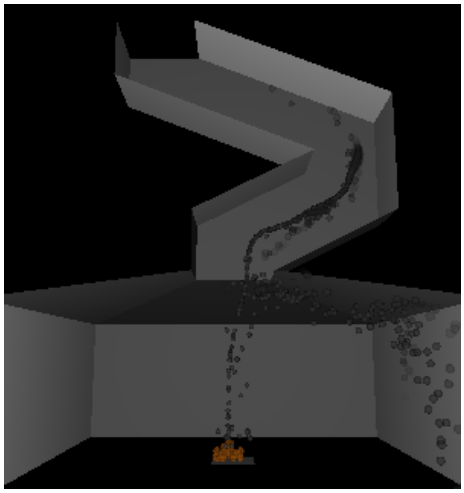


Figure 5.9: *The realistic looking motion field obtained by the algorithm described in section 5.5.1. The motion field is visualised by a set of semi translucent spheres that have been moved by the field. This picture is taken from the animation shown in figure 8.20 on page 135.*

the same time, the cooler air is warmed up and *dragged* into the jet). As the hotter parts of the gas rise more quickly than the regions that have mixed with the cooler air, more mixing and turbulence results. This is known as *thermal buoyancy*. The rising of hot gas also results in a change in *pressure* beneath the gas, causing the cooler air to rush in from the sides. Finally air moving slowly is pushed along by surrounding air, resulting in both velocity and temperature changes. This is known as *convection*. An overview of how these effects create motion is shown in figure 5.10.

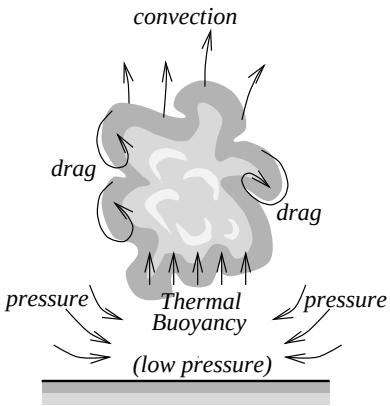


Figure 5.10: *The motion of a hot turbulent gas. The figure shows which effect contributes to which motion indicators when a hot gas interacts with a cooler environment. The rise of hot gas (thermal buoyancy) causes the cooler air above the gas to be pushed by the warm air currents (convection) while cold air rushes in below the rising gas due to changes in pressure. On the sides, where gasses of different temperatures meet, dragging causes the hot gas to slow while the cooler gas is accelerated.*

The algorithm models these effects along with solid object collision. It does not—however—consider effects caused by compression of gasses such as shock waves. This is due to the commonly used simplifying assumption that a gas is locally incompressible. It is not a simplification that weakens the algorithm when animating fire and smoke, as local compression in these cases has only a negligible effect on the overall motion, but it *is* a simplification that greatly reduces the mathematical complexity of the model. Another effect not modelled is internal diffusion caused by molecular motion of the gas. This is the effect that makes a gas keep moving even though conditions are calm. An accurate model of the molecular motion in a gas would be extremely complex.

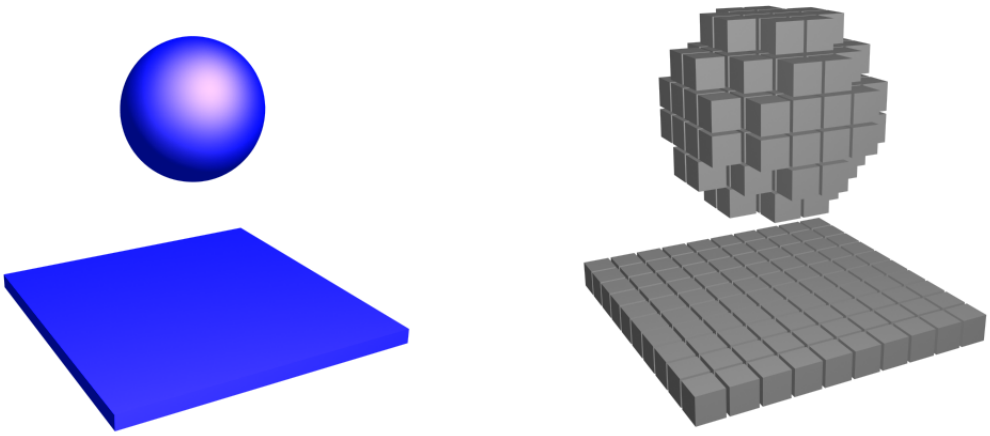


Figure 5.11: *A scene in Voxel Space.*

The above mentioned effects are handled by representing a volume of gas as a combination of a temperature field and a motion field. (The motion is represented as a 3D velocity-direction vector). To avoid having to calculate an exact solution throughout a 3D volume, the scene is represented in a voxel environment (as visualised in figure 5.11). Each voxel is identified by its position in the voxel environment, and contains information about the average temperature T in the volume covered by the voxel. The flow of gas from one voxel to a neighbouring voxel is recorded on the face between them. This is in fact a scalar value describing the velocity of the gas perpendicular to the face between the voxels. This is shown in figure 5.12. As the gas is considered to be incompressible, the pressure should be constant.

To calculate the motion field in the voxel environment the following steps are performed:

- Calculate the temperature in the scene, and use this to calculate thermal buoyancy.
- Add the effects caused by drag and convection
- Modify, by taking account of pressure effects. This is done by iteration.

In the following we shall give an account of how the above values can be calculated.

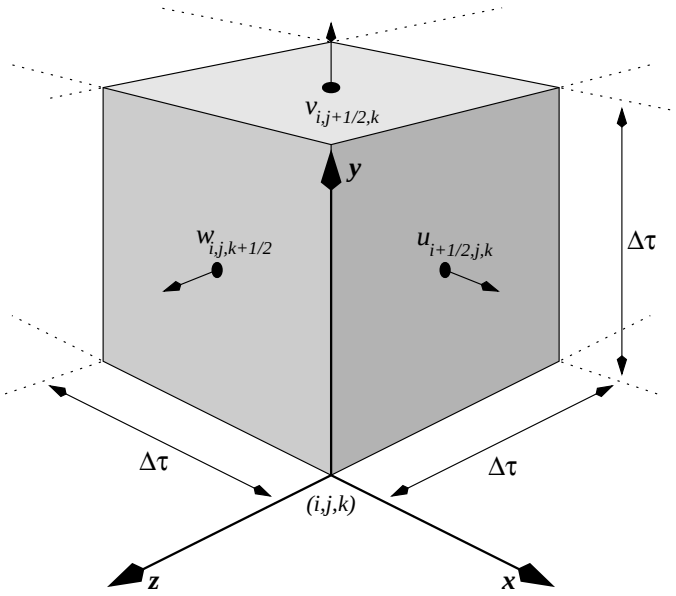


Figure 5.12: *The naming conventions of a voxel. This is voxel (i, j, k) in the voxel grid. u, v and w are velocities on the faces of the voxel. These are scalar values, describing the velocity perpendicular to the faces. Furthermore a voxel contains a value $T_{i,j,k}$ to represent the average temperature within the cell. $\Delta\tau$ is the side length of the voxel.*

The physics

To describe the forces acting within a gaseous phenomenon a set of equations called the Navier-Stokes equations have been developed. It is very difficult (if not impossible) to exactly describe these forces, as this would demand a model of the molecular interactions within the gas. To overcome this, the Navier-Stokes equations treats a gaseous phenomenon as a collection of small volumes of gas. These volumes must be so large that they contain enough molecules to make the behaviour predictable, while at the same time being so small that pressure, velocity, and temperature changes within the volumes are negligible. Assuming this, a set of differential equations can be set up to deal with the effects shown in figure 5.10. How they are obtained can be seen in many books on the subject, for instance [Tritton, 1988] or [Harlow and Welch, 1965].

These equations are very complex, and repeating them here in full would probably only serve to confuse the reader. Under the simplifying assumption that the gas is incompressible the equations are reduced to take a more manageable form, though. The Navier-Stokes equations say that the velocity change per time unit is just the sum of changes due to drag, convection, pressure, and thermal buoyancy. This is written:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla \cdot (\nabla \mathbf{u}) - (\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \frac{\partial \mathbf{F}_{bv}}{\partial t}, \tag{5.2}$$

where ∇ is the gradient operator, $\mathbf{u} = (u, v, w)^T$ is the velocity of the gas and p is the pressure of the gas. The term $\nu \nabla \cdot (\nabla \mathbf{u})$ describes the *drag* of the gas. The ν coefficient is called the *kinematic viscosity*. It describes the 'thickness' of the gas. Intuitively a small ν models a thin or light gas, where rotational motion is easily induced. On the other hand

a big ν models a 'thick' gas. The term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ describes changes due to *convection*, ∇p is the change due to differences in *pressure*, and $\mathbf{F}_{bv}$ is the *Thermal buoyancy*.

For those who are not comfortable with the ∇-operator, equation (5.2) is reformulated fully here:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) - (u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}) - \frac{\partial p}{\partial x} + \frac{\partial F_{bv,x}}{\partial t} \\
\frac{\partial v}{\partial t} &= \nu(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}) - (u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}) - \frac{\partial p}{\partial y} + \frac{\partial F_{bv,y}}{\partial t} \\
\frac{\partial w}{\partial t} &= \nu(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}) - (u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}) - \frac{\partial p}{\partial z} + \frac{\partial F_{bv,z}}{\partial t}.
\end{aligned}$$

Furthermore it is necessary to calculate how the temperature changes over time, to be able to calculate the thermal buoyancy. If the hot gas simply moves a temperature change occurs. This is a change due to *convection* of the gas. Otherwise the temperature changes as gas mixes. This is the change due to *diffusion* and *turbulence*.

As with equation (5.2) this can be written as:

$$\frac{\partial T}{\partial t} = \lambda \nabla \cdot (\nabla T) - \nabla \cdot T\mathbf{u}, \tag{5.3}$$

where T is the temperature of the gas and $\mathbf{u}$ is the velocity. The term $\lambda \nabla \cdot (\nabla T)$ describes the temperature change due to turbulence and diffusion (heating from nearby gas). λ is a scaling factor, representing both turbulent and molecular diffusion processes. $\nabla \cdot T\mathbf{u}$ is the term describing the convection of the gas.

If we reformulate this fully, we get:

$$\frac{\partial T}{\partial t} = \lambda(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}) - (\frac{\partial Tu}{\partial x} + \frac{\partial Tv}{\partial y} + \frac{\partial Tw}{\partial z}). \tag{5.4}$$

A very simple formula is used for the thermal buoyancy:

$$\mathbf{F}_{bv} = \beta \mathbf{g}_v (T_0 - T_k), \tag{5.5}$$

where $\mathbf{g}_v$ is gravity in the vertical direction, T_0 is the initial reference temperature (the temperature of the surrounding air) and T_k is the average temperature on the face between a voxel cell and the one above. β is the coefficient of *thermal expansion*, controlling the rate at which the gas rises—a bigger β models a gas that rises faster when heated. This formula is actually just a 'hack' to make the gas rise, and it just states that the hotter the gas is, the faster it rises.

Using equation (5.3) and equation (5.5) in (5.2), we end up with an animated motion field that should bring much more life to an animation than the static motion fields described in section 5.5. [Foster and Metaxas, 1997b] continue to show how this field is calculated numerically using the voxel environment.

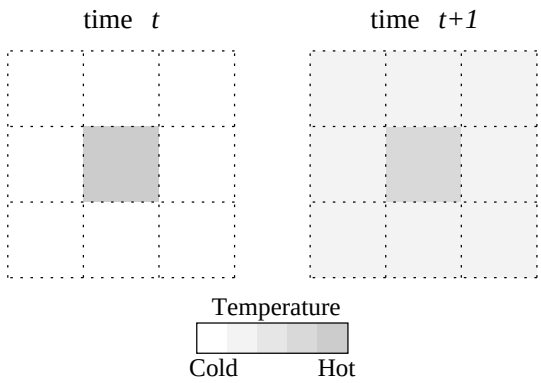


Figure 5.13: *Diffusion of the temperature field. The figure shows a collection of voxels of different temperature. The term* $\lambda(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2})$ *from equation (5.4) calculates the diffusion, so that the temperature is radiated from the hot voxels to the cold voxels. The temperature field is shown in two dimensions for clarity.*

Applying the equations to a voxel environment.

To solve the equations they are rewritten in a form that fits the voxel environment. As an example, a differential term such as:

$$\frac{\partial T}{\partial y}$$

could be approximated as a discrete counterpart by using a Taylor expansion. To simplify matters, let us assume that we work in a voxel environment where the voxels are cubic[8]. If $\Delta\tau$ is the side length of each voxel we get:

$$\frac{\partial T}{\partial y} = \frac{1}{2\Delta\tau}(T(y+\Delta\tau) - T(y-\Delta\tau)) + \mathcal{O}(\Delta\tau^2), \tag{5.6}$$

where $\mathcal{O}(\Delta\tau^2)$ denotes terms of second order or higher. In the same manner a second order derivative can be written as:

$$\frac{\partial^2 T}{\partial y^2} = \frac{1}{\Delta\tau^2}(T(y+\Delta\tau) - 2T(y) + T(y-\Delta\tau)) + \mathcal{O}(\Delta\tau^2), \tag{5.7}$$

If the terms of second order or higher are discarded, and the voxel environment notation is used we get:

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{1}{\Delta\tau^2}(T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k})$$

Let us look again at equation (5.4):

$$\frac{\partial T}{\partial t} = \lambda(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}) - (\frac{\partial Tu}{\partial x} + \frac{\partial Tv}{\partial y} + \frac{\partial Tw}{\partial z}),$$

This equation states that a change in temperature results from the changes caused by diffusion and convection. Figure 5.13 and 5.14 show how these effects change the temperature field in a discrete scene.

[8]The corresponding theory and formulas for voxel cells which are non-cubic can be found in [Foster and Metaxas, 1997a]. In [Watt and Watt, 1992] seven different types of scene subdivisions are proposed. Cubic voxels are called "Cartesian volume geometries".

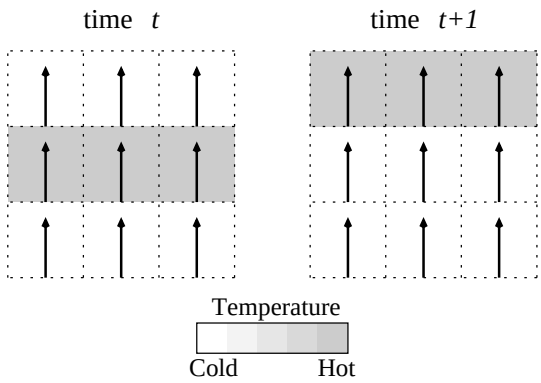


Figure 5.14: *Convection of the temperature field. This figure shows the effects caused by the convection term* $-(\frac{\partial Tu}{\partial x} + \frac{\partial Tv}{\partial y} + \frac{\partial Tw}{\partial z})$ *of equation (5.4). The arrows show the motion vectors in the field. Notice how the warm areas move as the gas moves. The temperature field is shown in two dimensions for clarity.*

To calculate the temperature changes of a discrete temperature field equations (5.6), (5.7) (and the similar equations for x and z) are inserted in (5.4), the terms of second order or higher are discarded, and the voxel notation is used. We arrive at:

$$\begin{aligned}
\frac{\Delta T^n_{i,j,k}}{\Delta t} &= \frac{\lambda}{\Delta \tau^2}\Big(T^n_{i+1,j,k} - 2T^n_{i,j,k} + T^n_{i-1,j,k} \\
&+ \quad T^n_{i,j+1,k} - 2T^n_{i,j,k} + T^n_{i,j-1,k} \\
&+ \quad T^n_{i,j,k+1} - 2T^n_{i,j,k} + T^n_{i,j,k-1}\Big) \\
&- \frac{1}{\Delta \tau}\Big((Tu)^n_{i+\frac{1}{2},j,k} - (Tu)^n_{i-\frac{1}{2},j,k} \\
&+ \quad (Tv)^n_{i,j+\frac{1}{2},k} - (Tv)^n_{i,j-\frac{1}{2},k} \\
&+ \quad (Tw)^n_{i,j,k+\frac{1}{2}} - (Tw)^n_{i,j,k-\frac{1}{2}}\Big)
\end{aligned} \tag{5.8}$$

where the suffix n means the given value at iteration number n. A term such as $(Tu)^n_{i+\frac{1}{2},j,k}$ represents the temperature flow between cells (i,j,k) and $(i+1,j,k)$ at iteration n, and is calculated as:

$$(Tu)^n_{i+\frac{1}{2},j,k} = \frac{1}{2}u^n_{i+\frac{1}{2},j,k}(T^n_{i,j,k} + T^n_{i+1,j,k}).$$

Using equation (5.8) the change of temperature over time in the scene can be calculated using an iterative scheme. Knowing the temperature and velocities in each voxel cell (i,j,k) at time t (iteration n), it is just a matter of applying the formula to get the values at time $t+\Delta t$ (iteration $n+1$), as:

$$T^{n+1}_{i,j,k} = T^n_{i,j,k} + \Delta t \frac{\Delta T^n_{i,j,k}}{\Delta t}.$$

This value is then directly used to calculate the thermal buoyancy (5.5) which in turn is part of the motion field. ($\frac{\partial \mathbf{F}_{bv}}{\partial t}$ can be found by subtracting $\mathbf{F}^n_{bv}$ from $\mathbf{F}^{n+1}_{bv}$).

Calculating the effects caused by drag and convection

We have now calculated the thermal buoyancy in the scene. The next step in the algorithm on page 62 is, to calculate the effects caused by drag and convection. From equation (5.2) it is noted that these effects can be written:

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

As with the temperature field, this is expanded into first and second order derivatives, resulting in three equations, one for each of the the u, v and w component of the motion field. This is what the equation for u looks like:

$$\frac{\partial u}{\partial t} = \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) - (u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}),$$

Again we approximate by inserting the values from the voxel grid. We get:

$$\begin{aligned}
\frac{\Delta u^n_{i-\frac{1}{2},j,k}}{\Delta t} &= \frac{\nu}{\Delta \tau^2}\Big(u^n_{i+\frac{1}{2},j,k} - 2u^n_{i-\frac{1}{2},j,k} + u^n_{i-\frac{3}{2},j,k} \\
&+ \quad u^n_{i-\frac{1}{2},j+1,k} - 2u^n_{i-\frac{1}{2},j,k} + u^n_{i-\frac{1}{2},j-1,k} \\
&+ \quad u^n_{i-\frac{1}{2},j,k+1} - 2u^n_{i-\frac{1}{2},j,k} + u^n_{i-\frac{1}{2},j,k-1} \Big) \\
&- \frac{1}{\Delta \tau}\Big((u^n_{i,j,k})^2 - (u^n_{i-1,j,k})^2 \\
&+ \quad (uv)^n_{i-\frac{1}{2},j+\frac{1}{2},k} - (uv)^n_{i-\frac{1}{2},j-\frac{1}{2},k} \\
&+ \quad (uw)^n_{i-\frac{1}{2},j,k+\frac{1}{2}} - (uw)^n_{i-\frac{1}{2},j,k-\frac{1}{2}} \Big)
\end{aligned} \tag{5.9}$$

where a term such as $(uv)^n_{i-\frac{1}{2},j-\frac{1}{2},k}$ must be read as $(u^n_{i-\frac{1}{2},j-\frac{1}{2},k})(v^n_{i-\frac{1}{2},j-\frac{1}{2},k})$. To find the u, v and w values at other positions than the centre of the voxel faces, simple interpolation is used. (Remember still, that a term such as u^n means u at iteration n).

With these equations we are now able to apply effects caused by drag and convection to our motion field calculations in the voxel environment. Examples of what this could look like can be seen in figures 5.15 and 5.16.

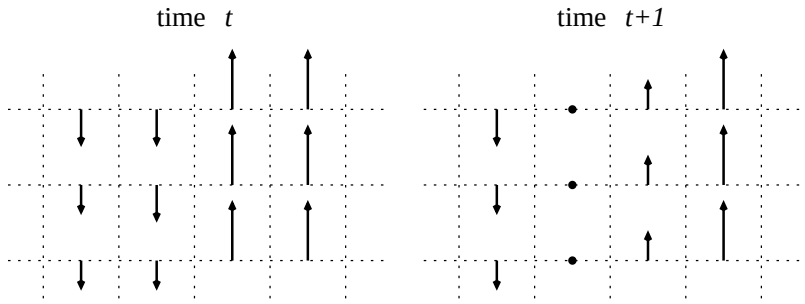


Figure 5.15: *Drag in the motion field. The fast upwards moving gas is slowed down at the edges when encountering the downwards moving gas. This effect is calculated by the term $\nu \nabla \cdot (\nabla \mathbf{u})$ from equation 5.2. The ν parameter describes the 'thickness' of the gas. It can also be thought of as how 'sticky' the gas is as a bigger ν increases the amount of drag, but at the same time makes it slower to change. The motion field is shown in two dimensions for clarity.*

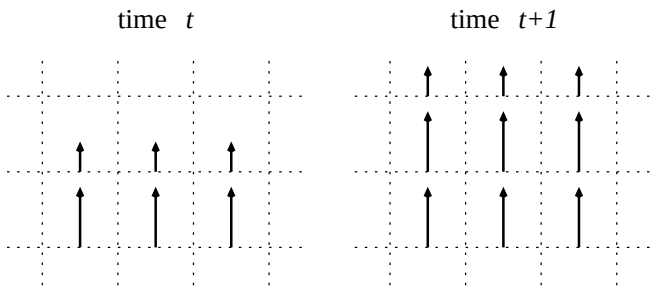


Figure 5.16: *Convection of the motion field. The effects caused when gasses moving at different speeds meet and push each other along. This is calculated by the term* $-(\mathbf{u}\cdot\nabla)\mathbf{u}$ *from equation 5.2. The motion field is shown in two dimensions for clarity.*

Calculating the pressure gradient and ensuring accuracy

The formulas (5.8) and (5.9) both suffer from the inaccuracy caused by discarding the terms of second order or higher in the Taylor series. In the case of the temperature field, this inaccuracy is not a big problem. It just results in some parts of the gas getting the wrong temperature, possibly causing the thermal buoyancy to be a bit wrong. However, for the motion field it means that the amount of gas entering a voxel might differ from the amount leaving. In the worst case this could lead to a scene where one voxel simply sucks away all the surrounding air. This could also be interpreted as mass disappearing from the scene.

Obviously the pressure of any voxel should be reflected by the difference in gas entering and leaving. In fact the pressure of any voxel should be constant, as we assumed that the gas in the scene is incompressible. Therefore a simple constraint to the calculation of the motion field should be that the amount of gas entering and leaving a voxel should be the same, or mathematically speaking:

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

Again we approximate using the Taylor series method, and rewrite the left hand side in terms of the voxel grid variables:

$$\begin{aligned}(\nabla \cdot \mathbf{u})_{i,j,k} &= -\frac{1}{\Delta\tau}\Big(u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k} \\ &+ \quad v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k} \\ &+ \quad w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}\Big)\end{aligned} \tag{5.10}$$

where $(\nabla \cdot \mathbf{u})_{i,j,k}$ is the *mass divergence* at the centre of voxel (i, j, k). This must be zero for all voxels in the grid. In the literature this is known as solving the classic three dimensional Poisson equation. [Harlow and Welch, 1965] produced one of the first numerical solutions in print, by defining a potential field ψ representing the relative discrepancy in mass between adjacent cells. For a given frame in the animation, this field is initially set to zero everywhere. Then ψ is iterated:

$$\begin{aligned}
\psi^{h+1}_{i,j,k} &= \frac{1}{4}\Big(\Delta\tau^2(\nabla\cdot\mathbf{u})_{i,j,k} \\
&+ \quad \psi^h_{i+1,j,k} + \psi^h_{i-1,j,k} \\
&+ \quad \psi^h_{i,j+1,k} + \psi^h_{i,j-1,k} \\
&+ \quad \psi^h_{i,j,k+1} + \psi^h_{i,j,k-1}\Big) \\
&- \psi^h_{i,j,k}.
\end{aligned} \tag{5.11}$$

This iteration is said to converge, when for all voxels:

$$\left|\frac{|\psi^{h+1}_{i,j,k}| - |\psi^h_{i,j,k}|}{|\psi^{h+1}_{i,j,k}| + |\psi^h_{i,j,k}|}\right| < \epsilon.$$

Now $\mathbf{u}$ is updated by subtracting the gradient $\nabla\psi$. We mentioned earlier, how a loss of mass could be thought of as a change in pressure. In fact it can be shown that $\nabla\psi = \nabla p$ [Harlow and Welch, 1965], so in theory we end up with a solution that ensures both the accuracy of the calculation *and* calculates the effects on the motion field due to changes in the pressure field:

$$\begin{aligned}
u'_{i-\frac{1}{2},j,k} &= u_{i-\frac{1}{2},j,k} - \frac{\psi_{i,j,k} - \psi_{i-1,j,k}}{\Delta\tau}, \\
v'_{i,j-\frac{1}{2},k} &= v_{i,j-\frac{1}{2},k} - \frac{\psi_{i,j,k} - \psi_{i,j-1,k}}{\Delta\tau}, \\
w'_{i,j,k-\frac{1}{2}} &= w_{i,j,k-\frac{1}{2}} - \frac{\psi_{i,j,k} - \psi_{i,j,k-1}}{\Delta\tau},
\end{aligned}$$

where u', v', and w' should be more accurate motion vectors than u, v, and w.

Unfortunately we have not been able to make formula (5.11) converge, even for very simple configurations. Foster and Metaxas have acknowledged the problems, and suggested us a modification to the algorithm, that should help [Foster, 1998]. Unfortunately we cannot get this to work either. Luckily our problems are solved by [Foster and Metaxas, 1997a], wherein they propose a simpler, more stable, but also slower method of calculating the effects caused by the pressure field.

The mass divergence $(\nabla\cdot\mathbf{u})_{i,j,k}$ is still calculated for every voxel (equation (5.10)). A positive mass divergence represents an influx of gas, which in the real world would correspond to an increase in pressure and subsequent increase in gas outflow. So by using $(\nabla\cdot\mathbf{u})_{i,j,k}$, we calculate the change in pressure of that voxel, and then use this pressure change to modify the motion vectors on the faces of the voxel. Of course this also modifies the motion on the faces of neighbouring voxels as the faces are shared, so the result will not be correct. It will, however, be a better approximation of the correct motion, if the adjustments are chosen intelligently. Therefore an iterative scheme can give us any desired level of accuracy.

When the voxel cells are cubic—as we have assumed— with side length $\Delta\tau$, the change in pressure for a cell is:

$$\delta p = \frac{\beta_0\Delta\tau}{6}(\nabla\cdot\mathbf{u})_{i,j,k} \tag{5.12}$$

where β_0 is a so called *relaxation coefficient*. This coefficient determines how much the motion field is updated according to the change in pressure δp. A value of 1 results in a change, that makes this voxel have a zero pressure change. A value greater than one results in an over-modification (*over-relaxation*). This can in some cases reduce the number of iterations required to reach an accurate solution, but it also introduces the possibility of divergence. In our implementations any value of β_0 greater than one almost always makes the motion field diverge.

When δp is calculated, the voxel cell face velocities are then updated:

$$\begin{aligned}
u'_{i+\frac{1}{2},j,k} &= u_{i+\frac{1}{2},j,k} + \delta p, \\
u'_{i-\frac{1}{2},j,k} &= u_{i-\frac{1}{2},j,k} - \delta p, \\
v'_{i,j+\frac{1}{2},k} &= v_{i,j+\frac{1}{2},k} + \delta p, \\
v'_{i,j-\frac{1}{2},k} &= v_{i,j-\frac{1}{2},k} - \delta p, \\
w'_{i,j,k+\frac{1}{2}} &= w_{i,j,k+\frac{1}{2}} + \delta p, \\
w'_{i,j,k-\frac{1}{2}} &= w_{i,j,k-\frac{1}{2}} - \delta p,
\end{aligned} \tag{5.13}$$

In figure 5.17 the effects of applying this iterative scheme is shown. The figure shows how it would look in the two dimensional case.

The algorithm repeated.

So, with all the above mentioned calculations, the algorithm sketched out on page 62 for calculating the motion of a hot, turbulent gas for the next frame in an animation, can now be more fully formulated:

1. Calculate the thermal buoyancy (5.5) using equation (5.8).
2. Calculate a temporary motion field (5.2) using equation (5.9). Use the just calculated thermal buoyancy ignoring the pressure gradient.
3. Calculate the mass divergence $(\nabla \cdot \mathbf{u})_{i,j,k}$ at each voxel cell using equation (5.10). If the mass divergence is 'too big' continue to 4. Otherwise the motion field is accurate.
4. Calculate an improved motion field by applying $(\nabla \cdot \mathbf{u})_{i,j,k}$ to equations (5.12) and (5.13). Go to 3.

The mass divergence is 'too big' if it exceeds some prescribed ϵ for any voxel cell. An ϵ of 0.0001 has been used in the example pictures of this thesis.

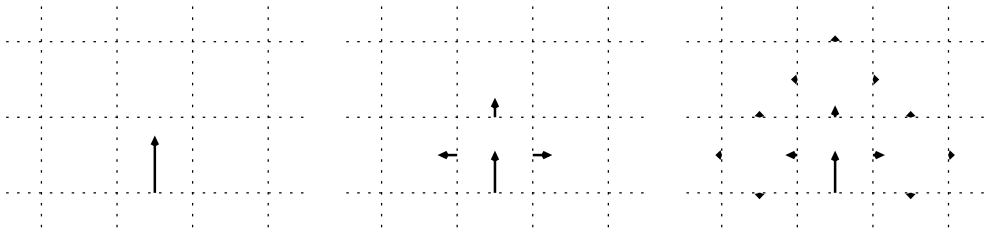


Figure 5.17: *A simple example of iterating the motion field calculations. The example is drawn in two dimensions for clarity.*

5.5.2 Interaction between gas and solid objects.

The algorithm described in section 5.5.1 can be modified to deal with interaction between gas and solid objects in the scene. First, the scene is reconstructed in voxel space. This leaves us with two kinds of voxels: Solid voxels and non-solid (gaseous) voxels. As an example look at figure 5.11, where the gaseous voxels are not shown.

Let us start by assuming, that the solid objects in the scene are non moving (moving objects will be discussed on page 74). In that case it should be clear, that the velocity perpendicular to a solid voxel face should be zero, as no gas should be able to enter or leave that voxel. This works if no part of the gas in the scene can move so fast, that it passes through one or more voxels per time step. In that case the interpolation of velocity vectors enables the gas to move through the solid voxel, as shown in figure 5.18. In any case such high velocities should be avoided, as the algorithm only uses neighbouring voxels in the calculations. Therefore the algorithm is imprecise (and actually unstable) if gas travels more than one voxel lenght per time step. This problem is easily solved by choosing a smaller time step and skipping rendering of frames accordingly, as the time step is used as a scaling factor in the calculations.

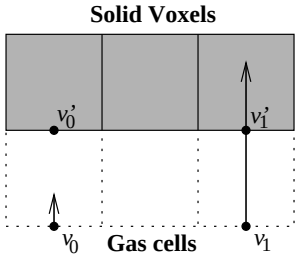


Figure 5.18: *Motion vectors for solid voxels. On the faces between solid voxels and gaseous voxels the velocity is set to zero (v_0' and v_1'). v_0 and v_1 are the caluculated motion vectors for the gas cells. v_1 is far too big. Interpolation between v_1 and v_1' would result in particles ending up* inside *the solid voxel.*

Let us return to the discussion of solid voxels. As mentioned the velocities on faces between solid and gaseous voxels should be kept at zero. This automatically has the nice effect, that the second term in equation (5.3) - describing the temperature change due to convection - becomes zero. The change in temperature in a solid voxel then only depends on the heat exchange from neighbouring voxels, as it should. Alternately the velocity between a solid and a gaseous voxel could be set to something besides zero. In that case, the solid would either emit gas (providing a way to model a gas jet) or suck gas (which could be used to model an open window).

The velocity on faces between *solid* voxels could also be set to zero, but other values are just as valid depending on which behaviour is wanted near the surface of a solid object. Consider the configurations of figure 5.19. In the example to the left, the velocity u_0' between the two solid voxels is chosen to be exactly the same as the velocity u_0. This models a perfectly smooth surface, where the gas is not slowed down when moving along the surface, such as glass or steel. In the middle example u_1' is set to zero, causing gas close to the surface to slow down, and thereby creating a small amount of turbulence. This could be compared with a concrete wall. In the rightmost example u_2' is set to $-u_2$, causing the

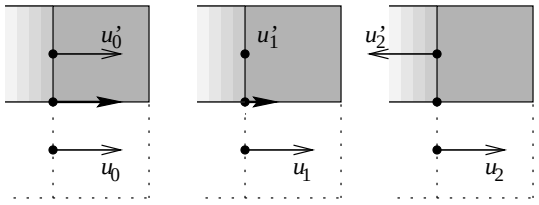


Figure 5.19: *Motion along the surface of solid voxels. The u and u' vectors are interpolated on the face between the solid and the gaseous voxel. In the example on the left, this interpolation gives the effect of a completely smooth surface. In the middle example the interpolation causes the gas to slow down, when nearing the surface. In the rightmost example the interpolation causes the gas to make a complete stop, giving the effect of a very rough surface.*

gas to make a complete stop at the surface. This models a very rough surface that creates a lot of turbulence.

The temperature of a solid object is set according to the temperature the object would have in real life. For instance if the scene shows a hot plate on a stove, the temperature of the voxels defined by the hot plate, should be set to the temperature of the hot plate itself. The temperature of gaseous voxels should be set to a chosen *background temperature* such as 25° C.

5.5.3 The usefulness of the model.

The biggest problem with defining a motion field by covering the scene with a three dimensional array is that turbulence on a smaller scale than the array grid size can not be represented. This means, that the quality of the results depends solely on the resolution of the array. But so does the complexity of the algorithm and the general storage use. Just representing a $100 \times 100 \times 100$ array of velocities (u, v and w) and temperature requires 16 MBytes of storage, assuming each voxel entry takes up 4 bytes. And to make matters even worse, the visualisation of smoke and fire in the scene might require that the motion of a given volume of air can be backtracked through time in n steps (as will be discussed in section 6.9.3), thus requiring $16n$ MBytes. Clearly the resolution of the motion field could not be much finer than this without running into major storage problems, but even a resolution of $100 \times 100 \times 100$ might be too coarse. Consider a visualisation of someone trying to burn down Saint Paul's Cathedral. This particular building is 108.7 metres high[9], so to contain it in the model the voxel size would have to be more than 1 m^3. It is clear that this resolution is too coarse to produce a convincing result if looking at a single burning instead of the whole chapel.

So to be able to get good results with the model it is necessary that the scale of the model is pretty much the scale of the resulting picture. Alternatively the voxel environment could be applied on just the part of the scene that is visible in the output (or just a bit more, to get accuracy at the borders), but in both cases zooming in the animation causes problems. If a really flexible solution is desired, it would perhaps be a good idea to try to make an adaptive subdivision of a scene into voxel space. A simple algorithm would be to always either double or bisect the voxel side length. When bisecting the side length, each

[9]and was finished in 1087! We have been unable to find out if that is a coincidence, but anyway, this is not a masters thesis on culture and history.

voxel is subdivided into eight new voxels. The temperature, pressure and wind field values of which can simply be found by interpolation. When doubling the side length, groups of eight voxels should blend together to form one voxel. As before, the temperature, pressure and wind field can simply be found by interpolation. If the subdivision should occur locally in the scene (thus ending up with voxels of different size) a modified *octree* could be used. Octrees are described in [Foley et al., 1990] and [Watt and Watt, 1992].

The only real problem with the adaptive subdivision would be to decide which values to apply on the borders of the voxel environment, and to decide when to change scale. For instance, if the scale is calculated by examining which objects are visible in each frame, a visible surface approaching the projection reference point could suddenly take up the entire viewport, and thereby change the scale. When this close surface then moves away detail has been lost. The simplest practical way to solve this problem would be, to let the user decide when the scale change takes place. In our examples (see section 8.4) we only use a 'once and for all' division of a scene into voxel space. This is done because a static subdivision *is* simpler to implement, and still shows how applicable the method is, as long as the amount of zooming in the animation is kept to a minimum. In the examples voxel grid resolutions of $45 \times 28 \times 45$ and $15 \times 9 \times 15$ have been used. Considering the above discussion this seems a bit coarse, but the addition of second level turbulence (as discussed earlier) helps to create pleasing results. The effect of this is clearly seen in figure 8.17 on page 132, where the same picture is rendered with and without addition of second level turbulence.

Voxel grid alignment.

Another problem is that it makes a big difference how the voxel grid is aligned. Consider figure 5.20. The figure shows two cases, where the alignment of the voxel grid causes undesirable results. In the example on the left it is clear that even though the particle

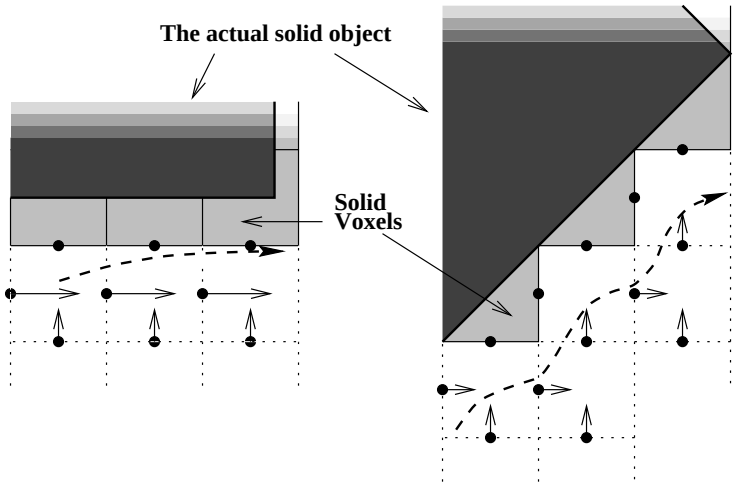


Figure 5.20: *Two examples of a voxel environment created by a solid object. The dashed line represents a particle's path through the motion field. The difference in alignment of the voxel field with respect to the solid object, gives a more turbulent particle path.*

moves in a convincing way, it never reaches the surface of the actual solid object. This happens because of the coarseness of the voxel environment. This is not a *big* problem—though. Because, as we shall see in section 6.9, it can be advantageous to visualise particles as small volumes of air, so the gap between the particle and the solid object will be filled. The other problem is far worse. In the example on the right it is shown how the particle's path along a solid object becomes far more turbulent, just because the object is no longer aligned parallel to the voxel grid. As it is, the model offers no solution to this problem other than using a more detailed voxel grid, but as the scene in figure 5.9 (page 61) shows, the effects caused by a bad alignment of the voxel grid are very difficult to spot.

Moving the turbulence field.

In section 4.1 it was seen that to make distinct features of the turbulence appear to move upwards in the animation, it was necessary to move the entire turbulence field. In the method described by [Stam, 1991] (section 5.5) second level turbulence was added to a static motion field. This would give exactly the same problem with distinct features staying in the same place in the animation. Stam then claims that the turbulence field can be moved by the static motion field. This works in theory, but it is difficult to see, how one should actually move the turbulence matrix according to the motion field, unless the motion field is constant everywhere. Stam offers no real solution to the problem, except to just move the turbulence field upwards by an amount that makes the animation look good.

When using an animated motion field (that is, the field obtained by using the algorithm on page 70) this problem is reduced, as the motion field itself contains the big level turbulent motion. In this way the distinct features of a turbulent motion *will* move correctly in the animation. In this model, the small scale turbulence is not necessary to give the animation a turbulent look, but is only used to add detail. Therefore it is not as important to move it correctly. Employing Stam's hack and making the second level turbulence rise slowly, should be good enough for generating convincing results. As the method can not calculate turbulence on a lower scale than the voxel size, the second level turbulence should be scaled to add detail at voxel size and smaller.

Movable solid objects.

A question that arises when evaluating the usefulness of the model, is if it allows solid objects to move. In the static motion field method proposed by Stam this would require that the animator should manually update the motion field at each frame of the animation. Of course this could be accomplished by defining a standard motion field around each movable solid object, which is then moved and transformed along with the object.

With the animated motion field method, the needed motion field changes are actually automatically calculated. The only thing that needs to be changed from the algorithm is that the velocity vectors on the faces of a solid voxel must reflect the motion of the actual solid object, as shown in figure 5.21.

The velocity vectors on the solid voxels pushes particles away in front of the object and sucks them in from behind the object due to low pressure, just as we would have expected. The only problem arises whenever the solid object enters or leaves a voxel. Suddenly that voxel changes state from gaseous to solid (or vice versa). Hopefully all particles have

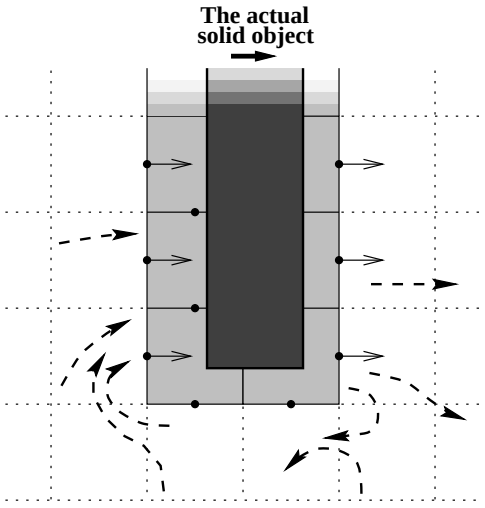


Figure 5.21: *A movable solid object in voxel space. The velocity of the object is reflected by the velocity on the faces of the solid voxels. The dashed lines show, how particles are being moved around by the resulting motion field. To simplify the figure, this field is not shown.*

already been pushed away from this voxel, but anyway it *will* probably create small 'jumps' in the gas velocity. It is difficult to say in advance whether these 'jumps' will be noticeable in an animation.

It should be clear that if the solid object moves more than one voxel per time step, the solution will be wrong. Again this should be easy to correct, by choosing a smaller time step. Movable solid objects have not been implemented in our raytracer.

Speed considerations.

One of the greatest advantages of the static motion field model is that the motion of any given particle is found by a simple interpolation, and no work need to be done to update the motion field between frames in an animation. If this is combined with a fast visualisation technique (for instance by using scan-line rendering [Foley et al., 1990]—possibly boosted by hardware) it allows for real time previews while modelling the scene.

When using an animated motion field, the field needs to be updated each frame. We recall that this update was mainly done by iteration, so the entire field is in fact re-calculated n times (with n normally ranging somewhere between 5 and 20 according to [Foster and Metaxas, 1997b]). On today's computers this is too slow to facilitate real time previews unless the motion field resolution is very small.

One might think, that this makes the job of animating the scene harder for the person setting up the animation, but we must remember that the whole idea of an animated motion field is to *automatically* calculate the motion of smoke and fire, and thus remove the need for an animator. If the field is well calculated, there will be no need to ensure that the animation looks correct, unless unnatural exaggerated effects are wanted. Whether it looks good or not is another question, which alone relies on personal preferences. Previews will still be needed to ensure that the fire parameters are set up correctly, so that the fire is not extinguished or evolve too drastically, and the smoke has the desired thickness. For this information it is not essential that the previews are real-time, as long as they are endurably fast.

Animated motion field vs. static motion field.

When comparing the usability of the static motion field with the animated motion field it should also be noted that some effects are simply much easier to create with one of the two algorithms than with the other. For instance, a helicopter flying through a dust cloud should make the dust twirl around the rotor blades. This effect should be very easy to create using Stam's static motion field, as a twirling motion field could be created once and for all, and then fitted around the helicopter for each frame of animation. An animated motion field would have to handle extremely fast moving solid objects at a very high voxel resolution to create the same effect.

On the other hand, the animated motion field automatically handles the direct and indirect spread of fire (see section 3.2.3) through the calculation of the temperature field. Stam's static motion field can not handle this without modifications.

When Stam uses his static motion field, he adds a contribution from the precalculated turbulence field described in section 5.4, to bring an fine turbulent movement to the particles. Obviously this can also be done when using the animated motion field as an animation tool. Here, though, a little care must be taken to avoid particles ending up inside solid voxels. Actually there is really no need to add turbulence to the motion of the particles. As we will show in section 6.9, the particles can be visualised in a way, that shows the turbulence of the wind field surrounding the particles. This is also an algorithm proposed by [Stam, 1991] (with inspiration from [Tritton, 1988]). What we intend to do, is to add some turbulence to the animated motion field during the visualisation process. This way, the animation of the particles is kept simple, while the visual output is turbulent and detailed. To our knowledge this has never been tried before, and interesting results are to be expected. More about that in section 6.9.

Related work

The kind of calculations shown in this section is generally called *Computational Fluid Dynamics* (CFD). More work has been caaried out along the same lines by [Baum et al., 1997] and [Mell et al., 1996]. These models feature a much more detailed incorporation of the physical and chemical processes involved than the model shown in this section, and very impressive results are obtained. A description of these models require a good knowledge of chemistry, and is therefore outside the scope of this thesis, but they seem ideal for future study. [Tritton, 1988] would be a good introduction.

Chapter 6

Visualisation

"It crawls! It eats you alive"

"The Blob"

A good fire model does not guarantee good results. In fact, the model might be as good as any model can get, but if the visualisation techniques are badly chosen the result might still be unusable. In this section a lot of ways of visualising fire are discussed, ranging from simple (but sometimes fast) techniques to very complex ones. In figure 1.6 on page 9 we have given an overview of how the visualisation methods are usually connected to the animation models described in the previous chapter.

This connection is sometimes so close that the choice of a model dictates the choice of a visualisation technique. Our raytracer implementation uses particle systems and motion fields as animation tools, and as shall be seen in section 6.9 blobs are ideal for visualising fire and smoke, making an implementation of blobs mandatory. In section 6.10 different ways of illuminating the world are discussed, and a simple radiosity algorithm is presented, which has also been implemented in our raytracer.

6.1 Scanline rendering

Scanline rendering (as described in [Foley et al., 1990]) is usually the fastest way of visualising models represented in three dimensions, and is the most used visualisation method today. There are some limitations to the rendering technique though, as reflection and refraction effects are very difficult to obtain. Another problem arises with the introduction of semi-transparent materials and objects, since the visualisation of these objects demands much more than "which polygon is the closest one to the camera concerning this pixel?". This problem can be even more complicated when objects intersect. Usually the Z-buffer is adequate, but, alas, it cannot for instance handle the situation of three semi-transparent polygons covering the same pixel. To solve this [Carpenter, 1984] developed the A-buffer. It is a simple array, representing each pixel of the frame buffer with a sorted list of the polygons that intersect for each pixel. The problem with the A-buffer is that it is slow and uses a lot of memory. This problem may be solved in the future, but for the time being it is not feasible. Another problem of the scanline rendering is the results made so far. All the attempts of implementing scanline rendering end up in a tradeoff between polygons and speed. This, of course, degrades the result.

The rendering methods of two dimensional fire certainly belong here. It would be overkill to start raytracing these.

The best three dimensional (or $2\frac{1}{2}$D really) results that use scanline rendering are obtained by using a technique which first renders the entire three dimensional scene and then *adds* effects like fire corresponding to the Z-buffer. These results will be discussed in this chapter.

6.2 Raytracing

Raytracing (as described in [Watkins et al., 1992]) with its enormous capacities can handle almost any type of natural phenomenon, given the time to do so. The problem with

raytracing is that it consumes a lot of time because of its computational complexity. If a computer graphics engine using raytracing should run decently real-time, it would have to use a large system of parallel processors.

Since the real-time aspect is of no concern to us, we have chosen to implement three dimensional fire in a raytracer. It ensures us that we can deal with all the aspects of digital fire without having to think of how to circumvent the problems that arise from scanline rendering.

6.3 Colour representation and restrictions

The usage of black body radiation points in the direction that the most used computer graphics colour representation model, RGB, is inadequate. The RGB model is very convenient since almost all computer displays today use RGB modulation; nevertheless it has several disadvantages. The RGB colour cube (see [Foley et al., 1990] for an explanation of it) is not able to represent all colours in the visible spectrum of light. Furthermore, if the internal representation of colours is also RGB, the results can only be linear combinations of the selected red, green and blue colours (which results in only mono-, duo-, or tri-chromatic colours), which eliminates usage of true black body radiation, which is polychromatic.

To solve this, one should use a model of the entire visible electromagnetic radiation spectrum (see [Clausen, 1998]). Demanding such correct chroma modelling is not without problems, since it poses some difficult choices because the spectrum is continuous which is impossible to model generally on a computer. A spline-based model could be used as long as the spectrum is simple. A radiance model of e.g. mercury (Hg), however, would be extremely difficult to model since it is a sum of numerous weighed delta-functions. To avoid this (or as a solution to the problem) the visual spectrum must be broken up into (equidistant) discrete bands. How these should be selected is hard to answer and the number of bands depends on the desired quality.

The usage of correct chroma modelling facilitates more correct rendering of radiation from non-monochromatic light sources, and reflection and radiosity of filtering mirrors of non-monochromatic light, amongst others. The downside of correct chroma modelling is the computational cost and memory usage (which is no longer a big problem due to cheap RAM and processor speedups, according to [Clausen, 1998]). We have chosen not to implement "correct" chroma modelling since a black body palette can be simulated using only monochromatic light.

The idea is very simple. After the palette is chosen—the black body palette might vary a little depending on the temperature of the phenomenon (ranges 1500-2000 Kelvin, see figure 3.6 on page 29)—a generated grey scale image is converted into a colour scale image using each pixel's grey scale value as an index into the black body palette. See the correspondence between grey scale and colour scale pixels in figure A.1 and the result of a conversion in figure A.2. This is—as already stated—an extremely quick hack, mostly used for real time rendered fire. The alternative is a simulation of real-life radiation and temperature, which would eliminate the possibility of real time rendering. The results from the hack are really good, so the choice of going for the radiation simulation should be the perfectionist's choice.

6.4 Polygons

Polygons are very crude and may not be ideal for visualising fire and smoke. since they can be rendered very fast, modelling fire using polygons has been attempted many times. The results have not been very good, however. It is most applicable in products where real-time rendering is important, such as real-time simulations, games and virtual reality.

Figure 6.1: *A polygon fire from Quake[Carmack et al., 1996]. Fast but not very fire-like.* †

For the computer graphics application "3D Studio v4.0" from Autodesk a plugin named FLAME.IXP [Schreiber et al., 1994] can be used to create flame polygons. The plugin offers many parameters to the fire (not all are mentioned here, see [Forcade, 1995] for a thorough explanation and previews): *flame* (flame tendril smoothness, life span frames, time start / end), *presets* (quiet candle, turbulent candle, bonfire, campfire, fireplace, torch, jet and rocket), *emitter* (flow type, strength, duration, particle scattering), *wind* (speed), *turbulence* (frequency and amplitude) and *whorl* (number of whorls, spin and drift). The idea is to create a set of key frames and then make the program make transitions between these key frames using 3D-morphing and then loop the entire set of generated 3D objects. A lot of key frames are needed which makes the job extremely tedious. The output is several slim polygon meshes, or more accurate: polygon strands, almost looking like waving seaweed or short curly fetuccini (sorry, but this is our impression). The top of the fire polygon meshes are square and not narrow in the ends as one might have expected. The fire emitter is crowded with bobbing polygon strands, which is not very fire like. There is no assignment of texture, so the user has to come up with this herself (this is not a simple task). If the fire object is video posted with a considerable amount of glow[1], the fire becomes fairly realistic only when viewed from quite some distance. If the duration of the loop is long, the looping motion is hard to spot, and if the viewing angle is changed over time, even harder. But, eventually, when the fire is animated and rendered, the result is poor.

We wonder if sampling of implicit surfaces can be used to model fire with polygons. We believe so. If a density function with parameters $\delta(x, y, z, t)$ is present, a better production of polygon fire (at least in motion and spread) than the above mentioned is possible. We

[1] Video post glow: The entire scene is rendered, and "glowing objects" are at pixel level added a colour value that corresponds to a silhouette of the object in the glow's colour convolved by a Gaussian filter.

have not dealt with this as it is quite a large project, and we therefore move this to our list of our future projects. We believe that the major advantage of using implicitly sampled surfaces is that the amount of sampling points can be fixed, set according to the distance to the fire and/or selected locally according to curvature.

6.5 Texture mapping

Visualisation of fire using texture mapped polygons has so far been applied for very limited purposes. In [Bukowski and Séquin, 1997] a fire was rendered inside an office building using simple texture maps to visualise the spread of fire, not to give a realistic visualisation of the fire itself. (See figure 1.1, page 5) Texture mapped fire mostly occurs when a real-time rendering is needed. There is a problem with this kind of rendering concerning both the looks of the fire (as it is very preview-like) and the lighting of the scene. Light sources other than point sources are costly and therefore conflict with the real-time issue.

Texture mapping is used in 3D first-person-shooter games like Unreal (a Quake clone, a snapshot from this game is shown in figure 1.4 on page 6). An animation of fire is mapped onto polygon which always "faces" the spectator (it rotates on the y-axis when the viewer moves around it). Any type of 2D rendered fire with an opacity map can be used (demo fire as described in section 4.1, 2D blob fire and even projections from three dimensionally modelled fires.). Alternately, a sequence of pictures of a real fire can be used with great results. It is fairly quick to render but has a major drawback when viewed from above. It becomes either almost invisible or looks very odd as the fire starts to move sideways instead of upwards.

For the second version of 3D Studio Max[Kinetix, 1996] a plugin called "Elemental Fire" is available in the texture assigning dialog. The plugin is quite simple as it takes three colours (black, red, and yellow are a good start, fading from yellow in the bottom of the object through red to black in the top—approximating a black body palette), a random seed and the amount of chaos as input parameters. The only problems with it is that it is a bit dull and it has no automated opacity map options, which has to be created as well and the result of *that* is not good as it only looks good on a black background, while all other choices of background give poor results as you get a black outline. It can be applied to all kinds of objects and all disadvantages mentioned above must be taken into account here as well.

6.6 Fractals

Fractal fire rendering has been attempted many times—with both poor and good results. The idea is most often to find an area in a fractal set and colour it with a black body palette giving the impression of fire. In [Barnsley, 1993] very shortly a fire rendering is described; the result is very fractal looking, and it is most possibly inanimate.

Knowledge on this part of fire rendering is hard to obtain, especially when it comes to animating fractal fire. Apart from the simple two dimensional fire created in section 4.1 using fractal noise [Watt and Watt, 1992], the only source of explained fractal fire we have found is in a book on creation of procedural textures [Ebert et al., 1998]. We have read the related parts of the book, and then contacted the authors in vain; the only material they placed at our disposal was virtually unreadable due to technical details. But all the images

that we have been able to find are very nice to look at and the animations are very good also (even though it is only smoke and clouds that are featured). The conclusion on fractal fire must be, for the time being, that it is most likely possible to model fire and smoke with fractals. How this is done we must add to our list of future research. Still—here are two of the images (figure 6.2) that we have found at [Musgrave, 1993]. Both feature the black body palette.

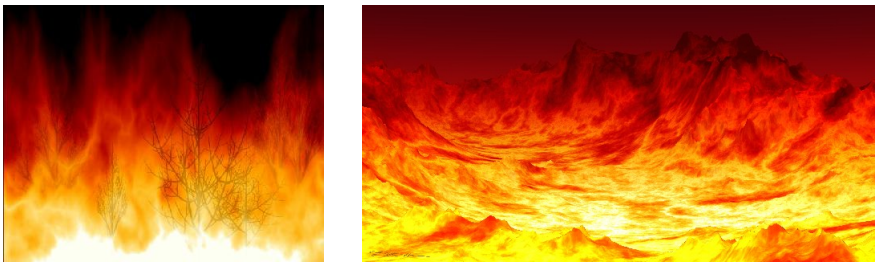


Figure 6.2: *A fractal fire and a fractal hell, from [Musgrave, 1993].* †

6.7 Voxels

The *vo*lume pi*xel* is briefly discussed in [Foley et al., 1990] under the term "spatial occupancy enumeration". [Watt and Watt, 1992] has a more thorough discussion. Voxels are generally used for three purposes. Initially they were derived from objects scanned spatially (as they are in *CT*[2]*-scanning*) so that the sampled object could be segmented and visualised. Second, voxels can also be used for mathematical models such as CFD as seen in section 5.5.1. A third is pure visualisation. [Kajiya, 1989] uses voxels to model fur and hair, so the application for modelling fire should be accompanied by a visualisation using voxels.

A large disadvantage of voxels is that up to n^3 voxels are needed to represent an object which has a resolution of n in each of its three dimensions. For a large n this has the undesirable effect of a huge memory consumption and computational time, whereas a small n might result in very poor accuracy. An example of an undesirable resolution for visualisation can be seen in figure 5.11 on page 62. Fire and smoke are phenomena that take up a lot of space due to the spread of fire and advection of the smoke.

Another problem concerns the visualisation properties of cubic voxels as it is only possible to make an accurate approximation of objects with surfaces parallel to the ones of the voxels' and with vertices on the voxel grid.

The visualisation of voxels is deeply connected to volume rendering. Voxels are often visualised as small volumes of gas, which are commonly used to model light shining through a *participating media*. Inside each voxel the properties of the participating media is set to be constant. With this approximation it is possible to find an exact solution for the scattering occurring when shooting a ray through a voxel as the voxels have a very simple geometry (often cubic) [Rushmeier and Torrance, 1987].

[2]CT = Computer Tomography.

Just before finishing this thesis we have found the works of the American firm Newtek (see [Newtek, 1999]). They have released a program called HyperVoxel, which is an add-on for the commercial visualisation application Lightwave3D. We have not had the opportunity to use HyperVoxel, but it does seem very impressive as it features animation and volume rendering of many phenomena: dust, water, clouds, explosions, and even some kinds of fire (see figure 6.3). The motion of the gaseous phenomena in the examples seems very much like the motion described by the Navier-Stokes equations, and judging by the explosions shown it appears that the application also features compressible volumes of gas. It is furthermore quite evident that the visual usage of voxels have been combined with some sort of texturing function.

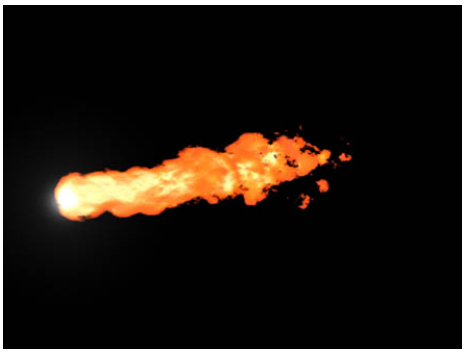


Figure 6.3: *A 'burning' comet made by 'Hypervoxel', an example from [Newtek, 1999].*

6.8 Particle systems

A particle system is in itself not a visualisation tool, it is merely a modelling tool. However there are several different very simple way to visualise a particle system. Lines were used as a particle visualisation tool for the Genesis bomb descussed in chapter 1 and section 5.2.2. Particle systems can be visualised using any form of object. Lets take a closer look at them.

6.8.1 Dots, line segments, and the like

How dots, line segments, and polygons should be used when visualising particle systems might seem simple, but some considerations must be made and the applicability of each type of primitive is important. Also many of the simple objects raise questions concerning blending objects at sub-pixel level, which poses the usual problems with anti-aliasing and texturing. Due to an expectation of poor results we have chosen not to implement visualisation of fire using these primitives and thus have not dealt with these problems. But other more complicated means of visualising particles can lead to very good results. Especially *blobs* and *bitmap splatting*, which are discussed below. These visualisation methods *have* been implemented. Blob visualisation is used in our raytracer, while we have also created a separate program for visualising bitmap splatting.

Dots are small objects, that are needed in large numbers if they are to be noticed. Dots are fabulous for creating very simple and very fast visualisations of a particle system, but the visual quality will not be high.

Line segments are somewhat better than dots but only to a certain extent. A line extends the dimensionality of the point which could be perceived as a motion blurred point (e.g. a motion blurred spark from a bonfire). The line segment does not have to be straight, but can also be a described by several straight line segments derived from speed and acceleration or even described by a spline. A line segment cannot be textured, but might be coloured differently along the segment either using a colour function or table. The same is the case concerning transparency.

Triangles and squares have been used in the early days to visualise fire with very poor results, since their shape does not resemble any of the physical characteristics of fire. The usual way of visualising these kinds of simple polygons is that the normal of the polygon always points towards the camera. The usage of squares have lately experianced a renaissance due to its immense rendering speed, as we soon shall see in the section on *bitmap splatting* below.

Spheres and cubes are applicable to a certain extent. The result is not in any way fire like whether they are shaded or texture mapped, but spheres can at least be used for fast previews of blobs. Blobs, which are spherical objects, are described later, in section 6.9.

Polygons are a bit more interesting because they are not necessarily planar. The example of the snow flake from section 5.2 is a good one. The problem with this kind of primitive is that it demands a simple description of the object that is to be visualised. So what we have to describe is local visual characteristicss of fire and then make a polygon representation of this. So far we have not seen any polygon representations that are even close to convincing, and we dare not think of the effort needed to get it.

A more complex kind of polygon particle can be an object which is a set of polygons or merely a three dimensional object. But as already stated: we have not implemented any of these, so we do not have any empirically obtained experience with them. Nevertheless we doubt that the results are better than any of the other visualisation methods mentioned.

Blobs are ideal visualisation primitives for particle systems describing a gaseous phenomenon, as blobs are volumetric objects capable of being distorted by wind currents. This makes their capability of rendering fire very good. It is this type of primitive we will concentrate on in this thesis, in the thorough description in section 6.9.

Particle systems can also consist of particle systems themselves (they are still not a visualisation method, just a modelling and animation tool). Normally different types of behaviour are wanted for each level of particle system, again the snow flake is a good example. [Paramount, 1982] is an example of a particle emitter emits particle systems (see figure 1.3 on page 6) as earlier stated in section 5.2.2.

Bitmap splatting [Laur and Hanrahan, 1991] introduce bitmap splatting as a technique for simulating volume rendering. The idea is simply this: Create a *"splat"* footprint (see figure 6.4), which is a bitmap (possibly with a fire like appearance) with an opacity map. The figure shows six different splat footprintss. Above the bitmaps a three dimensional visualisation of the splats are shown, where the height of the objects

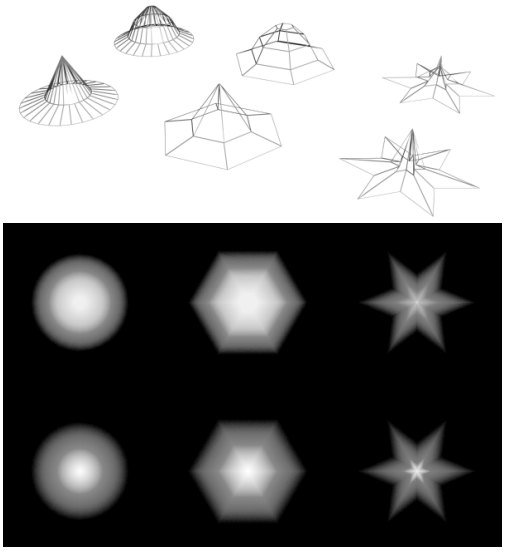


Figure 6.4: *Different splat shapes and their footprints. The splat shapes simply specify the intensity of the footprints (small 2D bitmaps).*

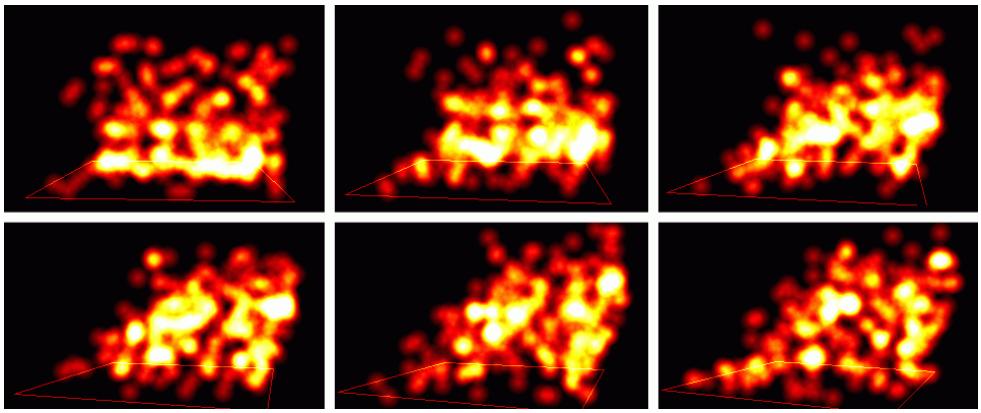


Figure 6.5: *Six frames from the real time bitmap pasting 3D fire.* †

correspond to the intensity of the splat footprint. A splat is used as a visualisation of a particle. The splats' characteristics may vary spatially (they can be shaded individually) and temporally (due to animation constraints). Then these bitmaps are pasted onto the image canvas, usually back to front. In this way quite accurate volume renderings can be made of quite complex objects (i.e. a motor block or CT-scans, see [Laur and Hanrahan, 1991]), including fire and smoke. This rendering technique is quite fast as most graphics adapters today supports bitmap pasting. We have made a 3D real time fire using this technique with a simple "splat", see figure 6.5 for a few frames. The result is promising; the fact that the fire is three dimensional is quickly noticed. This method can be improved if each "splat" is advected locally according to the surrounding turbulence field, using the advection technique from section 4.1.1.

6.9 Blobs

When describing more complex forms to be visualised, such as raindrops or the animation of soft materials, problems arise when the surfaces are intersecting or just deformed. How should an animator animate the intersection of two water droplets slowly bouncing into each other? The sphere primitive is not suitable for this purpose. Instead, the blob, introduced by [Blinn, 1982b] should be used. Blinn initially used the blob to visualise the distribution of electrons in a density cloud of a covalent bond, but it is also very convenient when rendering gaseous forms. In this section we will concentrate on its applicability on animating and visualising fire and smoke.

6.9.1 "What is a blob?"

A blob[3] is a location $\mathbf{c}$ surrounded by a density field described by a density function, δ. It can be visualised in two ways: as a depiction of the density field or by selecting a field threshold, which makes it possible to define inside and outside of the blob, and in this way describe the shape of the blob. When we visualise fire and smoke we use the first of these two, as the examples in chapter 8 show.

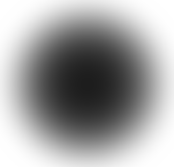


Figure 6.6: *A blob - a location and a field description.*

The δ function could be defined as in [Blinn, 1982b]

$$\delta(r) = be^{-ar^2} = e^{-ar^2+\ln(b)}$$

where r is the distance from the centre of the blob $\mathbf{c}$ to a given point in space $\mathbf{x}$. The b and a are constants that alter the "blobbiness" of the field. The a component controls at what rate the density decreases as the radius increases. A high a results in a fast exponential decrease of density, while an a of zero gives a constant density of the component b—which can be interpreted as a scaling parameter or as the mass of the blob particle. The b component can be varied over time to achieve a global change in density. Figure 6.7 shows this δ function for the values $b = 1$ and $a = 0.7$.

Blinn's definition of the δ function is a bit impractical when it comes to visualising the blobs, as the contribution never reaches zero. Furthermore, calculating the exponential function is slow. [Wyvill et al., 1986] created a δ function as a sixth degree polynomial:

[3]Blinn used the description "Blobbiness" to describe the shape of his particles, hence the name "blobs", but they are also referred to as "meta-balls".

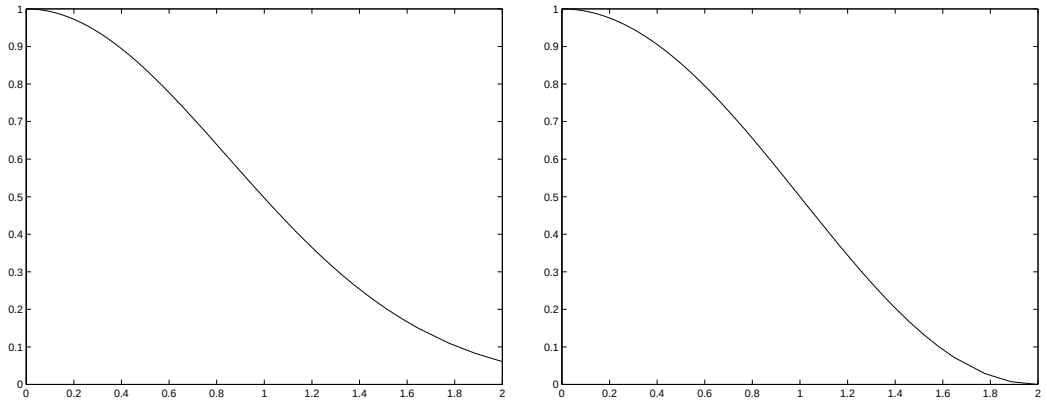


Figure 6.7: *Blinn's and Wyvill's blob density functions. On the left is Blinn's function:* $\delta(r) = be^{-ar^2}$*, for* $b = 1$ *and* $a = 0.7$*. The function on the right is Wyvill's polynomial:* $\delta(r) = -(\frac{4}{9})r^6/R^6 + (\frac{17}{9})r^4/R^4 - (\frac{22}{9})r^2/R^2 + 1$*, for* $R = 2$*.*

$$\delta(r) = \begin{cases} -(\frac{4}{9})r^6/R^6 + (\frac{17}{9})r^4/R^4 - (\frac{22}{9})r^2/R^2 + 1 & \text{if } 0 \leq r \leq R, \\ 0 & \text{if } R < r \end{cases}$$

where R is the distance from the centre at which the blob is no longer visible.

This function has the desirable properties: $\delta(0) = 1, \delta(R) = 0, \delta'(0) = 0, \delta'(R) = 0$ and $\delta(R/2) = \frac{1}{2}$, thus ensuring a smooth density field if blobs overlap. This δ function is also shown in figure 6.7. Note the similarity between the two functions.

When visualising a set of intersecting blobs their added fields and—if desired—a boundary threshold describe the shape of the blob object. If we associate a mass m_i for each of N blobs $\mathbf{c}_1(t), \ldots, \mathbf{c}_N(t)$ we define the basic blob object density as:

$$\rho(\mathbf{x}, t) \quad = \quad \sum_{i=1}^{N} m_i(t)\delta(|\mathbf{x} - \mathbf{c}_i(t)|). \tag{6.1}$$

The blob density is another way of giving the probability distribution for a particle being placed in $\mathbf{x}$. It is defined as $\frac{\rho(\mathbf{x},t)}{M}$, where M is the total mass at time t.

6.9.2 Diffusion and advection of blobs

Without distortion of the blobs, the blob object will look quite regular and too nice due to the lack of a turbulent surface and interior. A distortion of the blob is therefore essential. Advection-diffusion of blobs can be done by using an algorithm like the following (exerpt from [Stam, 1995], where also new blob births are handled).

```
for each time count do
     for each new blob do
          find its origin and size
     for all blobs do
          advect centre of blob
          increase size of blob
          kill blob if
               its density function contribution becomes
               insignificant
```

This algorithm, however, does have the problem that the contribution of a blob becomes insignificant if the blob becomes too big. This is due to the mass of the blob remaining constant while the blob size increases. Thereby the density drops. This makes the above mentioned diffusion awkward, but it could be solved by splitting up the blob—when it becomes larger than a certain threshold—into several smaller blobs and reconstructing the blob from these (normally a fixed number of blobs is used). Another approach is blob back warping.

6.9.3 Blob Back Warping

When a blob is moved through a turbulent wind field it should no longer resemble a spherical object, but be warped by the field. It is difficult (if not impossible) to represent the appearance of a warped blob directly, so another approach is needed.

Imagine a position in space $\mathbf{x}$, where the density of a warped blob is desired. imagine also that we only know the location of the blob centre $\mathbf{c}$, the radius r, and the wind field that has distorted the blob $\mathbf{F}_{\text{ext}}$. This is the setup in the left part of figure 6.8. The blob density function is not known for the warped blob, so the density in $\mathbf{x}$ can not be found directly. If we assume that the blob was unwarped some time ago (Δt), we can move $\mathbf{x}$ and $\mathbf{c}$ backwards through the wind field $\mathbf{F}_{\text{ext}}$, and find their respective position at that time, $\mathbf{c}_{-1}$ and $\mathbf{x}_{-1}$. (the right part of figure 6.8). These positions are then used in the blob desity function.

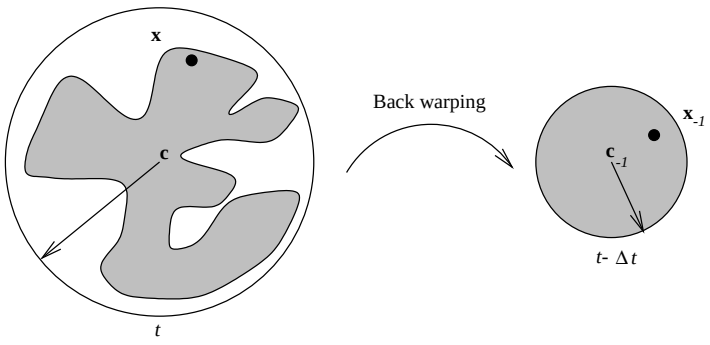


Figure 6.8: *Backwards Warping of a blob*

This method of warping points back through time is called the blob back warping algorithm [Stam, 1991]. Each point $\mathbf{x}$ is then tracked back through time by solving the following equation:

$$\mathbf{x}_{-1} = \mathbf{x} - \int_{t}^{t-\Delta t} \mathbf{F}_{\text{ext}}(\mathbf{x}(s), s) ds,$$

approximated by

$$\mathbf{x}_{-1} = \mathbf{x} - \Delta t \mathbf{F}_{\text{ext}}(\mathbf{x}, t - \Delta t)$$

where $\mathbf{x}(s)$ is the location of $\mathbf{x}$ at time s and $\mathbf{F}_{\text{ext}}(\mathbf{x}, s)$ is the wind field in position $\mathbf{x}$ at time s. The blob is assumed to be unwarped (spherical) at time $t - \Delta t$.

A more precise approximation to the back warping integral is obtained by taking N small backwarp steps instead of one big. The backwarping integral is then approximated by N discrete back warp steps, using the following iterative algorithm:

$$\mathbf{x}_{-(n+1)} = \mathbf{x}_{-n} - \frac{\Delta t}{N} \mathbf{F}_{\text{ext}}(\mathbf{x}_{-n}, t - n\frac{\Delta t}{N})$$

The number of iterative back warps has considerate impact on the visual characteristics of blobs. In chapter 8, we investigate the effects of varying the different parameters involved in back warping.

The choice of a turbulent wind field is very important when backwarping blobs to get good results. In [Stam, 1991] blobs are backwards warped through the precalculated turbulent wind field described in section 5.4. This has exactly the same drawback as using the precalculated turbulence as a particle animation tool (as discussed in section 5.5): There is no interaction between the gasses and the solid objects in the scene.

As an improvement we have decided to let the backwarping be performed using the animated motion field from section 5.5.1. To our knowledge this has never been done before, and it should hopefully shape the blobs around the solid objects in the scene. The biggest problem is that the animated motion field has a very low resolution, so the result will probably be too smooth to look like fire or smoke. To overcome this, we have chosen to add second level turbulence using the precalculated turbulent wind field, thus merging the methods of [Stam, 1991] and [Foster and Metaxas, 1997b]. The second level turbulence should only be strong enough to add details, but should not change the global appearance of the backwarped blobs. In section 8.5 a comparison between the two types of turbulence can be found.

6.9.4 The blob visualisation algorithm

Raytracing is very useful for visualising blobs. Blobs can be visualised in many ways. One way, is to show the blob density field as a volume of semi translucent gas *(smoke)*. An other way could be to depict a blob as a volume of glowing gas *(fire)*. It is even possible to visualise a blob as a refractive media *(water, caustics and shimmering effects)*. Raytracing facilitates all these different interpretations by sampling the rays through the blobs.

Figure 6.9 shows how two blobs are sampled along a ray. As each blob is sampled, a colour contribution is approximated along with the blob's translucency (or density)—the amount of light being absorbed by the blob. This approximation is achieved by dividing the blob in n intervals, sampling the blob colour and translucency in each centre, and then assuming these values to be constant in the interval.

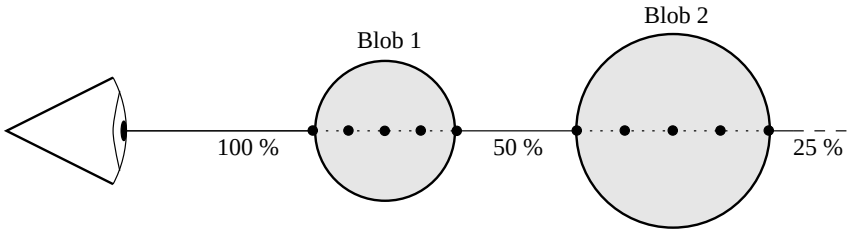


Figure 6.9: *Two blobs being sampled along a ray. Each blob is sampled using four sample steps. Each pair of dots in a blob constitutes a sample interval, where the blob is sampled in the middle.*

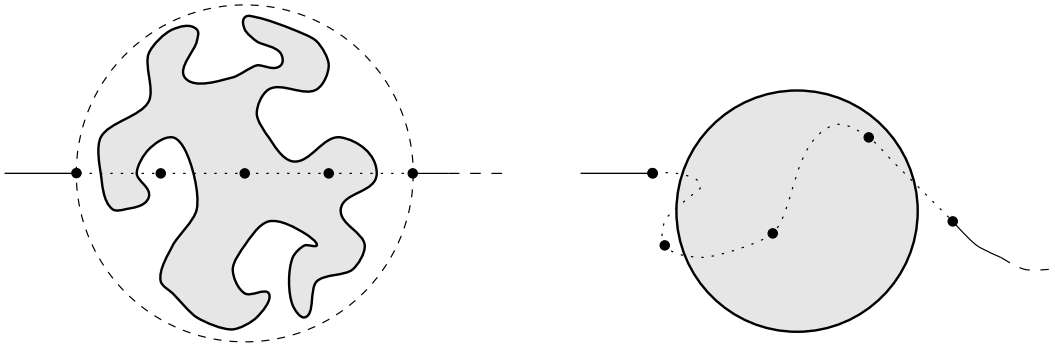


Figure 6.10: *This figure illustrates how the raytracer handles warped blobs. The picture on the left shows the ray intersecting a blob. The dashed sphere shows the maximum radius of the warped blob. In this example the blob is sampled in 4 sample intervals (5 sample points) along the ray. As the warped blob densities can not be found directly, the sample points are moved backwards through the wind field, thus in fact warping the ray instead of the blob, as shown in the figure on the right.*

As described in section 6.9.3, it is possible to warp blobs according to a wind field. It was noted that this was done by moving the sample points backwards through the field, and then using these 'backwarped' sample points to calculate the resulting blob density along the ray. This is visualised in figure 6.10.

Choosing the sample points

Some details must be considered when raytracing blobs by sampling the ray through the blob in n steps. Obviously this scheme implies that the ray is sampled at finer intervals in small blobs than in big blobs. This should be seen as a benefit, as a global sampling of the ray could be too coarse and miss small blobs, or it could be to fine and waste rendering time with non-noticeable details. On the other hand, it also means that the sample intervals might differ in length for the blobs in the scene.

Figure 6.9 showed a ray-tracing with 2 blobs, each occluding 50% of the light. The numbers along the ray shows the visibility of the objects in the corresponding points. This visibility is used as a scaling factor for the calculated colour contribution of the blobs. In this figure, choosing the sample points and calculating the translucency is straightforward.

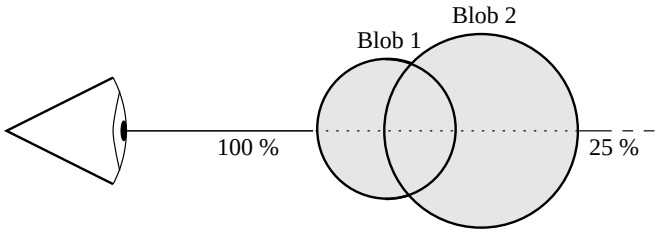


Figure 6.11: *Two overlapping blobs which are sampled along a ray. The visibility in the overlapping region is difficult to calculate.*

When using blobs modelling gas volumes, each ray will contain several overlapping blobs more often than not, as in Figure 6.11. In the overlapping area the total colour contribution and translucency of the blobs becomes much more difficult to calculate.

Ideally the basic blob object density function $\rho(\mathbf{x}, t)$ (as shown in equation (6.1), page 87) could be calculated analytically and then used on the whole blobby object. The whole set of overlapping blobs could then just be handled as if they were one big blob. Unfortunately $\rho(\mathbf{x}, t)$ can not be calculated when the blobs are back warped. Instead the blob object density could be sampled in the overlapping region, just as it was sampled in the regions containing only one blob. So, to visualise the set of blobs, the obvious procedure would be to divide the ray for each blob intersection, as shown in figure 6.12.

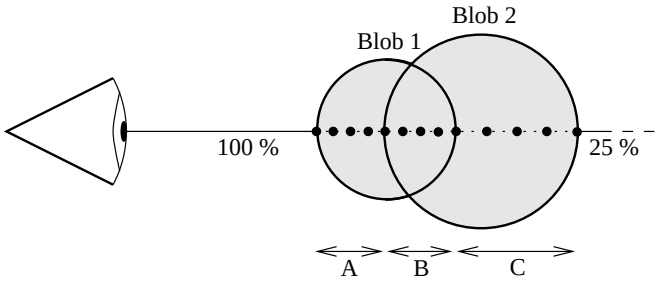


Figure 6.12: *Two overlapping blobs being sampled along a ray. The ray is split up in section A, B and C, where section B is traced for both blob 1 and blob 2.*

In this figure each region is sampled using the same amount of sampling steps as when sampling a single blob. This is the the simplest possible division imaginable, and it obviously results in too fine an interpolation. On the other hand this division makes it easy to know exactly which blobs contribute to the given sample point. Another approach could be to calculate the original interval division points for all overlapping blobs and then sample along these points, as is shown in figure 6.13.

This method requires some more housekeeping of which blobs contribute to which intervals, and where these intervals are. On the other hand, the number of sample intervals is halved, so in spite of the extra housekeeping overhead this method is probably faster.

The biggest problem with these two solutions is the complexity. In the worst case all blobs contribute to each sample interval. So if k is the number of blob sample steps and n is the number of blobs, the total number of sample intervals is kn (or $2kn$ with the simple division method). For each interval all blob contributions are calculated, thus ending up with a complexity of $\mathcal{O}(kn^2)$.

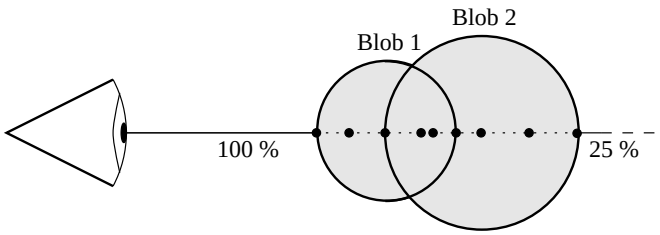


Figure 6.13: *Two overlapping blobs being sampled along a ray, as in figure 6.12. In this example, however, fewer sample intervals are used.*

One way to bring down the complexity of the algorithm would be to completely ignore the problem of overlapping blobs. Each blob is simply fully rendered when it is encountered along the ray - thus ending up with a complexity of $\mathcal{O}(kn)$. This method is straightforward to implement, but *can* result in visible artifacts if the intersecting blobs are of different types, as the nearest blob dominates the other blobs too much. This would be very clear in an animation if two blobs are almost completely overlapping, and the nearest blob moves just behind the first intersection of some other blob.

Luckily these artifacts are not noticeable as long as the intersecting blobs have properties that are nearly alike, as will be the case in a smoke cloud—or when the blobs are very translucent light emitters as in a flame. So in most cases the faster algorithm is good enough. Both the faster and the more precise algorithm have been implemented. The precise algorithm is only implemented in the 'simple' version (the one shown in figure 6.12), as it is a bit more precise than the version depicted in figure 6.13. Image quality and rendering times for these methods are compared in section 8.7.

6.10 Illumination

In computer graphics a lot of different ways of lighting a scene exist, ranging from the very fast hacks to the very complex, but also much more realistic models. The 'classic' way of lighting a scene uses point light sources and ambient lighting. A point light source is (as the name suggests) a light source that has no size. It is located at some position in space, and emits light in some colour. The light can be positioned infinitely far away in some direction, thereby functioning as a directional light source, or it can be restricted to emit light only in certain directions, thereby creating a spotlight effect. Point light sources are often combined with the usage of ambient light to make sure that all objects can be seen, even if they are located in the shadow cast by another object. The reader should be familiar with these different kinds of lighting, and the computational models involved. If not, consult any introductory book on computer graphics, e.g. [Foley et al., 1990].

So far we have not discussed ways of making the fire illuminate the scene. The choice is once again a choice of speed versus realism.

When speed is important the solution is to 'cheat' by not letting the fire emit any light on the surroundings. The whole scene is rendered normally, and then if there are any fire pixels which cover pixels in the resulting picture, a red or yellow colour tone can be added. Good results can be obtained this way at a low computational cost [Reeves, 1983]. If lighting of the surroundings is wanted, this can be simulated by using some sort of

pulsating light source.

More realistic results can be obtained by letting each fire particle consist of a point light source in addition to the above mentioned solution. This method can result in far more realistic looking scenery at the price of higher rendering time, while it still keeps the lighting model simple. Even more realism can be obtained by incorporating the inter-reflectivity between objects. In this section we will look closely at the *radiosity* light model, how it can be applied, and why it is useful when visualising the illumination of fire.

6.10.1 Secondary illumination

The simple model of a scene rendered with point light sources and an ambient background lighting actually has nothing to do with the laws of physics. For instance, no light sources are points (though the stars are close approximations, due to their distance), and no surfaces passively receive light. All surfaces reflect and emit light, though often to some very small extent. In the radiosity light model these effects are taken into account. This results in much more realistic shading. The downside is the high prerendering computational cost involved, but as the radiosity calculations are view independent, the price is only paid once in static scenes, where only the camera moves. This has made radiosity a success in recent action computer games, something one would have thought impossible a decade ago.

A scene involving fire can hardly be called static, so it will be necessary to recalculate the radiosity for each frame. The question is if there is any *particular* reason for implementing radiosity when animating fire with particle systems, other than the fact that the radiosity light model gives more realistic pictures overall? The answer must be that this depends on which role the fire particles have on the calculation of motion in the scene. In section 5.5 it is suggested to let the fire particles emit heat and in this way control the heating of the surroundings. As heat is just light with another wavelength, this is actually radiosity. Therefore it would seem obvious to include full radiosity calculations for all surfaces and light sources, and not just for fire particles.

Admittedly, we have not used heating from fire particles in our implementation. Section 5.5 moves on to describe a way of calculating gas motion, using the temperature of the burning surface itself and not the fire particles. So, in fact, the only real reason we have for implementing radiosity is that it just looks considerably better. Also one could argue that treating each fire particle as a light source would have approximately the same time complexity as using radiosity, while at the same time resulting in very sharp shadows, which is not what you would expect from a fire.

In the following section we will give an algorithm for computing radiosity. This follows closely the work of [Wallace et al., 1989]. The algorithm is called a 'progressive radiosity algorithm', and can be configured to produce fast or precise results - or many level in between. In [Wallace et al., 1989] comparisons are given to other algorithms.

6.10.2 A radiosity algorithm

When a surface is lighted by a light source, an amount of light (probably in some other colour) will be reflected by the surface to light up other surfaces. Some surfaces might also emit light all by themselves. This is the radiosity model. To render a picture with this light model, it is necessary to calculate the amount of energy (or light) each surface receives from each other surface. This is mainly done by finding the *form factors* in the

scene. The *form factor* from surface i to surface j is the fraction of energy leaving surface i that arrives at surface j and is written F_{ij}.
The radiosity of a surface is defined as the total amount of energy leaving the surface, and is given by:

$$B_i A_i = E_i A_i + \rho_i \sum_{j=1}^{n} B_j A_j F_{ji} \tag{6.2}$$

where

$$\begin{aligned} B_i &= \text{radiosity of surface } i \text{ (energy per unit area)} \\ A_i &= \text{area of surface } i \\ E_i &= \text{emitted energy per unit area of surface } i \\ \rho_i &= \text{reflectivity of surface } i \\ B_j &= \text{radiosity of surface } j \\ A_j &= \text{area of surface } j \\ F_{ji} &= \text{form factor from surface } j \text{ to surface } i \end{aligned}$$

The intuitive way to read this equation is, that the amount of energy radiated from a surface equals the amount of energy emitted plus the amount of incoming energy reflected.

When calculating radiosity in a scene the initial assumption is that no surfaces receive energy from any other surface - that is: The summation in (6.2) is zero. This makes the radiosity zero for all surfaces except light sources which emit light. The following steps are then repeated until the total amount of light in the scene converges.

- Select the surface with greatest radiosity (B_i).
- Compute form factors from that surface to all surface elements (i.e. vertices) in the environment.
- Based on these form factors, add the contribution from the source surface to the radiosity of each element.

This algorithm makes it clear that the real problem is finding the form factors. Let us look at a simple example. If we have two differential areas dA_1 and dA_2 and the configuration of figure 6.14, then the form factor from dA_1 to dA_2 is:

$$dF_{dA_1 \to dA_2} = \frac{\cos\theta_1 \cos\theta_2}{\pi r^2} dA_2 \tag{6.3}$$

where

$$\begin{aligned} \theta_1 &= \text{angle between normal at } dA_1 \text{ and direction to } dA_2 \\ \theta_2 &= \text{angle between normal at } dA_2 \text{ and direction to } dA_1 \\ r &= \text{distance between differential areas } dA_1 \text{ and } dA_2 \end{aligned}$$

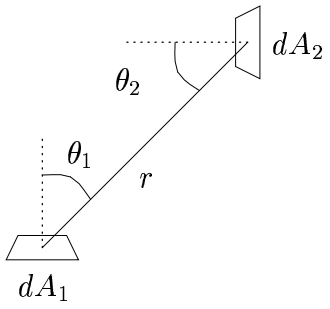


Figure 6.14: *The form factor from differential area dA_1 to differential area dA_2.*

To calculate the form factor from dA_1 to some *finite* area A_2 it is necessary to integrate the differential form factors. That is

$$F_{dA_1 \to A_2} = \int_{A_2} dF_{dA_1 \to dA_2} = \int_{A_2} \frac{\cos\theta_1 \cos\theta_2}{\pi r^2} \, dA_2$$

Note that r, θ_1 and θ_2 are different values for each differentail area dA_2 on A_2.

The straightforward way to reach a numerical solution to the above integral would be to divide surface A_2 into smaller regions of area ΔA_2 (see figure 6.15). If these areas are small enough equation (6.3) can be used to approximate a solution for each region, resulting in:

$$F_{dA_1 \to A_2} \approx \sum_{i=1}^{n} \frac{\cos\theta_{1_i} \cos\theta_{2_i}}{\pi r_i^2} \Delta A_2$$

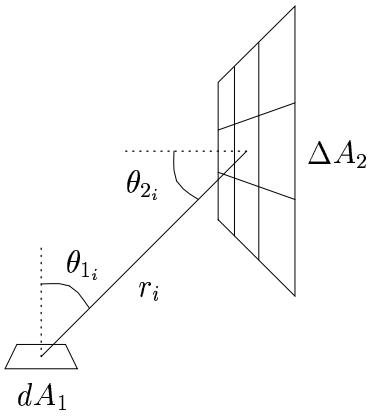


Figure 6.15: *The form factor from differential area dA_1 to a collection of Delta areas of size ΔA_2.*

This result is unfortunately very imprecise and prone to numerical errors. Especially the values tend to grow to infinity if ΔA_2 becomes larger than r_i; that is, if the two surfaces are big and close. So please forget we ever presented this method.

Another possibility is to approximate the surfaces ΔA_2 by geometric shapes so simple that the form factors can be derived analytically. One such simple shape is a disc. Consider

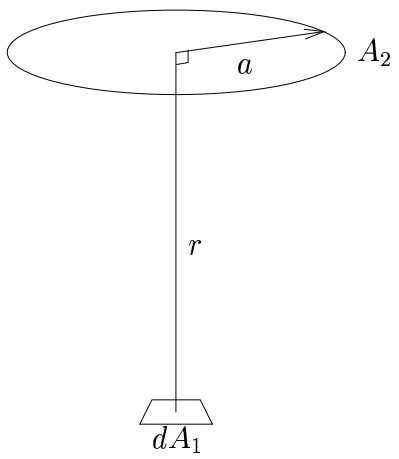


Figure 6.16: *The form factor from differential area dA_1 to a disc A_2.*

a region with differential area dA_1. We want to find the form factor from this region to a disc of area $A_2 = \pi a^2$. Consider the simple configuration of figure 6.16, where the normal of the disc points directly at the differential region and vice versa. If the distance between the regions is r, the form factor is:

$$F_{dA_1 \to A_2} = \frac{a^2}{(r^2 + a^2)}$$

This is the form factor from a differential region to a disc, but as the second step of the algorithm for computing radiosity specifies, we need the form factor from a surface to a vertex - a differential area. That is, we need to find $dF_{A_2 \to dA_1}$[4]. There is a principle called *the reciprocity principle* which states that if the surfaces A_1 and A_2 have the same area then $F_{A_1 \to A_2} = F_{A_2 \to A_1}$. As the form factor depends on the receiving surface's area the following holds for surfaces of different area:

$$\frac{F_{A_1 \to A_2}}{A_2} = \frac{F_{A_2 \to A_1}}{A_1}$$

Now let surface A_1 be a differential area again:

$$\frac{F_{dA_1 \to A_2}}{A_2} = \frac{dF_{A_2 \to dA_1}}{dA_1}$$

This brings us to the following formula for the form factor from A_2 to dA_1:

$$\begin{array}{rcccl} dF_{A_2 \to dA_1} & = & dA_1 \frac{F_{dA_1 \to A_2}}{A_2} & = & dA_1 \frac{a^2}{(r^2+a^2)A_2} \\ & = & dA_1 \frac{a^2}{(r^2+a^2)\pi a^2} & = & \frac{dA_1}{\pi r^2 + a^2 \pi} \\ & = & \frac{dA_1}{\pi r^2 + A_2} & & \end{array}$$

[4]The form factor from a surface to a differential area is a differential value. That is why it is written $dF_{A_2 \to dA_1}$ and *not* $F_{A_2 \to dA_1}$

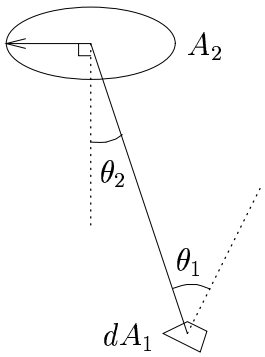


Figure 6.17: *The form factor from disc A_2 to differential area dA_1.*

In general the disc normal is not pointing at the differential area as is depicted on figure 6.17. As a general formula the following approximation can be used.

$$dF_{A_2 \to dA_1} \approx \frac{dA_1 \cos\theta_1 \cos\theta_2}{\pi r^2 + A_2} \tag{6.4}$$

This solution is robust even when the two surfaces are close because of the appearance of the area A_2 in the denominator of (6.4). So the earlier problem with values diverging as A_2 got bigger than r is solved.

It is possible to achieve a more correct solution by actually finding the form factors between non simple polygons. This is a problem, that was posed by Lambert in 1760, and solved in 1993 by [Schröder and Hanrahan, 1993] (a step of the way by [Baum et al., 1989]). The resulting formulas are more complex, and disc approximations are much simpler to work with as only one point and an area is used to approximate a disc. we have chosen to settle for the disc approximation.

We arrive at the following algorithm to calculate the form factor from a surface with area A_2 to a vertex. A vertex is a point in space and has no area. Therefore it can be considered as a differential area dA_1.

- Divide the surface into n delta areas. These areas are approximated by discs. The radius a of such a disc is of no importance as only the area is needed. This area is A_2/n. The delta areas are defined by a single sample point - the centre of the disc.
- Find the angles θ_{1_i} and θ_{2_i} (see figure 6.17) for the n delta areas.
- Calculate the total form factor as the sum of form factors from the n delta areas. This is done using

$$dF_{A_2 \to dA_1} = dA_1 \frac{1}{n} \sum_{i=1}^{n} \delta_i \frac{\cos\theta_{1_i} \cos\theta_{2_i}}{\pi r_i^2 + A_2/n} \tag{6.5}$$

where δ_i is the fraction of light reaching the sample point from the vertex. For instance $\delta_i = 1$ if sample point i is totally visible to the vertex, and 0 if totally occluded. The

reason for the $\frac{1}{n}$ scale that appears in the formula is, that each of the delta areas only emits/reflects $\frac{1}{n}$'th of the energy emitted/reflected by the whole surface.

The last problem is defining a value for dA_1 to use in equation (6.5). dA_1 is the area of the vertex, and should then logically be zero, which would make the total form factor zero. But luckily dA_1 can be ignored (that is: $dA_1 = 1$) when calculating the form factor as we shall shortly see.

As defined in equation (6.2) the radiosity of surface i due to energy received from source j was:

$$B_i A_i = \rho_i B_j A_j F_{ji}$$

By inserting the formula for our previously calculated form factor (6.5) we get

$$B_1 dA_1 = \rho_1 B_2 A_2 dA_1 \frac{1}{n} \sum_{i=1}^{n} \delta_i \frac{\cos\theta_{1_i} \cos\theta_{2_i}}{\pi r_i^2 + A_2/n}$$

where the differential area dA_1 cancels out. So—finally—the radiosity at vertex dA_1 due to illumination by source A_2 is:

$$B_1 = \rho_1 B_2 A_2 \frac{1}{n} \sum_{i=1}^{n} \delta_i \frac{\cos\theta_{1_i} \cos\theta_{2_i}}{\pi r_i^2 + A_2/n} \tag{6.6}$$

When the radiosity calculations are complete, each vertex in the scene will have received a light-contribution due to radiosity. This contribution can then be used for a Gouraud shading of the surfaces. This Gouraud shading can be performed independent of the viewpoint. Alternatively, the form factors could be used to perform Phong shading when the viewpoint is known. This does not seem like a profitable idea though, as the radiosity between surfaces is not likely to produce noticeable specular highlights. More information on Gouraud shading and Phong shading can be found in [Foley et al., 1990].

The way the algorithm has been described above, radiosity is only calculated in each vertex of a scene. This might give very poor results if the surfaces are big. To overcome this problem, each surface could be subdivided into any number of smaller surfaces. A quicker and more elegant way is to define a grid on each of the surfaces, and then use the intersections of the grid lines as if they were vertices. The resulting radiosity contributions can then be stored in the grid, instead of in the vertices. In this way we have created a kind of *light map* or *radiosity map*. This can be considered the same as a texture map, but instead of defining the colour in a certain position, it defines the lighting in that position. See figure 6.18.

6.10.3 Form factors between surfaces and blobs

As discussed in 6.10.1 we need to calculate the lighting of both surfaces and blobs based on the light emission of the blobs. This is accomplished by finding the form factors between all elements in the scene. Until now only surface to surface form factors have been discussed (equation (6.5)) but with a few simplifications blob to blob, blob to surface and surface to blob form factors can be approximated.

A blob is basically a warped sphere. This makes a precise calculation of the form factor a very difficult process. But as the blobs are small and numerous good results can

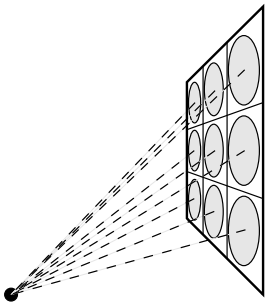


Figure 6.18: *The calculation of radiosity from a surface to a point. The surface is approximated by a grid of discs. Each disc receives and radiates a certain amount of light. This constitutes the* radiosity light map *on the surface.*

be obtained by approximating blobs with discs [Stam, 1995]. Consider figure 6.19. Here it is shown how the warped blob is approximated by a disc. As the figure shows, the approximation has the nice feature, that the discs normal points directly at the surface (that is $\theta_2 = 0$).

With this simple approximation it is possible to find the form factors between all types of objects in the scene. The results - which are generalisations of formula (6.5) - are shown in table 6.1.

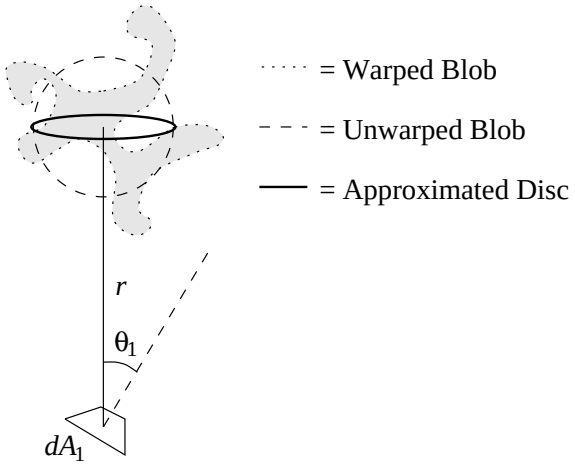


Figure 6.19: *A blob can be approximated by a disc, oriented with the normal pointing against the differential surface* dA_1.

$$
\begin{aligned}
\text{surface to surface} &: dF_{A_2 \to dA_1} = dA_1 \frac{1}{n} \sum_{i=1}^{n} \delta_i \frac{\cos\theta_{1_i} \cos\theta_{2_i}}{\pi r_i^2 + A_2/n} \\
\text{blob to surface} &: dF_{A_2 \to dA_1} = dA_1 \delta \frac{\cos\theta_1}{\pi r^2 + A_2} \\
\text{surface to blob} &: dF_{A_2 \to dA_1} = dA_1 \frac{1}{n} \sum_{i=1}^{n} \delta_i \frac{\cos\theta_{2_i}}{\pi r_i^2 + A_2/n} \\
\text{blob to blob} &: dF_{A_2 \to dA_1} = dA_1 \delta \frac{1}{\pi r^2 + A_2}
\end{aligned}
$$

Table 6.1: *The form factors between blobs and surfaces*

Chapter 7

Implementation

"I'm am the god of Hellfire
And I bring you fire!"

The Doors

Now we have dealt with most of the ways of animating fire and smoke. There is a small chance that these methods have been enhanced further, but we have not been able to obtain any knowledge of this. So what we need now is to implement some of these methods to examine their capabilities in practice.

We have experimented a great deal to try out the works of the authors of the mentioned methods. Some of those quickly turned out to be of little value, compared to those presented in this chapter. In other words: We think that these methods presented here are truly the best.

We have implemented all types of dimensionality, namely 2D, $2\frac{1}{2}$D and 3D. The two dimensional fire (described in section 7.1) is the advanced black body cellular automata fire and the black body turbulence advected fire presented in section 4.1. The $2\frac{1}{2}$D fire is an example of bitmap splatting, the additive bitmap pasting of a three dimensional particle system (section 6.8.1). The three dimensional fire (described in section 7.3) is the model proposed by Jos Stam, using a particle system (section 5.2), backwards warping (section 6.9.3) and the animation constraints told about by Foster and Metaxas (section 5.5.1).

7.1 2D fire — Cellular automata and turbulence advection

Cellular automata fire

We have implemented a real time version of the cellular automata fire. As described in section 4.1 this mainly consists of averaging, so we will not tire the reader with implementation details here.

The implementation features advanced video feedback, random advection of the grid, 'interesting' insertion of fire (from section 4.1.1), and different choices of black body palettes. The size of advection and amount of inserted fire can be modified at runtime. The choice of calculation of mean and interpolation method can not be changed without altering the program and recompiling.

The intriguing thing with this implementation is that it actually does look a lot like real fire. It even has the amazing feature that real fire also has: it makes one sit passively looking into the flames, captured by the randomness of its shape and colours.

The program listing is placed in appendix D.1. The figures 4.4 and A.4 show some screen shots.

Turbulence Advected Fire shape

Furthermore we have implemented the turbulent warping of a standard 2D fire shape as described in section 4.1.2. The turbulence function has been modified to include a time variable, and the program is set to create a sequence of subsequent images. Images from this sequence have been used as illustrations, for instance the images of figure 4.7. The program is listed in appendix D.2, and as can be seen it has been kept very primitive. At this time it is not possible to change any parameters without recompiling.

7.2 $2\frac{1}{2}$D fire — Bitmap splatting

We have made an example $2\frac{1}{2}$D fire, which too is kept simple. We have used a simple three dimensional particle system and constant sized, circular splats which in the blending gives a smooth looking fire. The fire is slowly rotated around the y-axis, so that it can be viewed from all sides. A small constant wind is used as the only external force, and the particles are advected (randomly) a little bit each frame. Example frames can be seen in figure 6.5, page 85.

This type of fire also has the power of paralysing the spectator, though not as good as the 2D cellular automata fire.

7.3 The TSR raytracer

We have implemented a raytracer, which creates fire-like objects using particle systems of blobs which move in a turbulent wind field. The TSR raytracer (short for Theo's & Søren's Raytracer) reads the scene from an input file, animates it any number of steps and outputs one or more image files as an output. TARGA has been chosen as the file format for the images because of this format's simplicity. The source code can be found in appendix D.4, and in figure 7.1 the most important parts of the raytracer are shown. To keep the figure simple a lot of functions have been omitted. These are mostly support functions used for special calculations, functions for linked list manipulations or class constructors and destructors.

In the following sections we will give a short overview of the functionality of the raytracer. This overview is presented to give the reader an idea of how the different areas of this thesis can be put together in a final design. Furthermore, we find it necessary to describe a few of the algorithms used, as this better enables us to make an analysis of the runtime complexity of the raytracer.

First we give a short introduction to the scene file format (section 7.3.1). This section shows how few parameters are needed to create a burning surface. Next we describe the radiosity algorithm (section 7.3.3), and finally the raytracing (section 7.3.4).

The raytracer uses the output of a the program (`fftурbu.cpp`) that creates a $16 \times 16 \times 16 \times 16$ turbulence array as described in section 5.4.1. This turbulence array is used in the visualisation of blobs, and can be exchanged with a new one by executing the helper program `fftурbu` again.

7.3.1 Parsing input scenes

We have implemented a parser to handle the reading of scene files. For this purpose we have created our own file format—the TSR file format. The reason for creating our own format is that this way we can concentrate on the features we want to deal with, and these features are presented in a way that easily fits into the data structures used in the raytracer. Another reason is to bring down the number of parameters needed to define a fire.

In this section we will give a short description of how to use the file format. We will only present the basic ideas. In appendix C.1 the full features of the file format is given. As an example, we will just show how simple it is to create a burning surface with

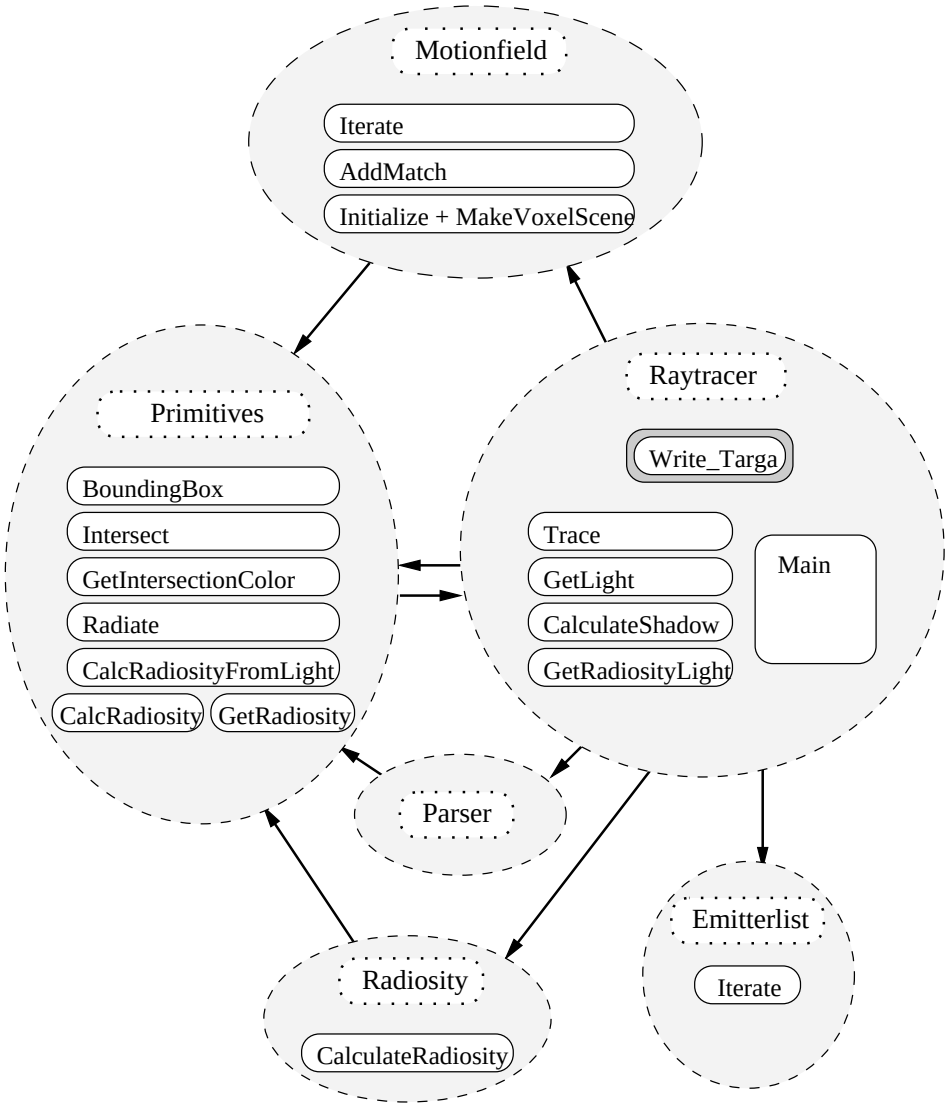


Figure 7.1: *Overview of the raytracer. The dashed gray ellipsoids represent different classes or logical parts of the program. The white boxes are functions or class methods. An arrow from one ellipsoid to another means, that the first logical part might use the other. The figure has been simplified. For the reader interested in a more detailed description of how the different functions call each other, a more detailed version of this figure is given in figure D.1 in appendix D.*

our implementation. This illustrates our success in minimising the amount of parameters needed to create fire.

The TSR file format uses a mixture of global state parameters and a stack inspired approach to define a scene. Parameters such as colour, temperature, radiosity parameters, texture mapping, and so forth are defined globally, and the values are valid for all following definitions of graphics primitives. A triangle is defined by three points (vertices), while blobs and spheres are defined by a center point and a radius. This should not surprise anybody, and we only mention it because we need to point out that these vertices must

be defined in advance, thus making them easy to reuse by letting several primitives share the same vertex. By letting the primitives be defined by shared vertices, it is easier and quicker to transform the scene and to implement rendering effects such as Gouraud or Phong shading. Actually none of these features have been implemented (although triangles have been *prepared* for Phong shading), but by using this design they should be easy to add if needed.

As the vertices are defined, they are given numbers starting with zero and then increasing for each new vertex. These numbers are used to identify each vertex when defining the primitives, after which time the vertices can be forgotten or used to define another primitive. If the vertices are no longer needed they can be removed. In this way, the numbering system can be thought of as a stack of vertices.

The file format uses ASCII text to describe the scene. For instance a simple scene is given by:

```
// light
color 0.8, 0.8, 0.8
light 150, 50, 100

// triangle
vertex   75, -75,  0
vertex  -75, -75,  0
vertex    0,  75,  0
triangle 0,1,2
```

This TSR file results in the picture shown figure 7.2. The scene uses a default camera positioned in $(0, 0, 100)$ looking at $(0, 0, 0)$.

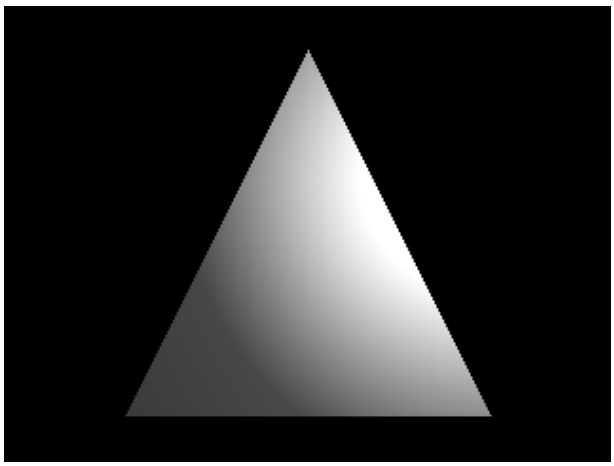


Figure 7.2: *A very simple scene.*

When a texture or fuel map is used, the texture map must first be loaded and texture coordinates must then be applied to the vertices of the texture mapped triangle. A texture coordinate (`tvertex`. Not to be confused with `vertex`) is just a position on a texture

map, where $(0,0)$ is the lower left corner and $(1,1)$ is the upper right. The command '`tritex i, j, k`' is used to define, that the following triangles will have the texture coordinates numbered `i`, `j`, and `k` mapped to its vertices. Figure 7.3 shows how a texture is applied to the triangle from the simple scene shown in figure 7.2. This is done by the following TSR file:

```
readtexture 1, Textures/texture_lenna.tga

// light
color 0.8, 0.8, 0.8
light 150, 50, 100

// texture map
texture 1
tvertex   1, 0
tvertex   0, 0
tvertex 0.5, 1
tritex 0,1,2

// triangle
vertex   75, -75,  0
vertex  -75, -75,  0
vertex    0,  75,  0
triangle 0,1,2
```

Figure 7.3: *A very simple texture mapped scene.*

A surface can be made combustible, and a lot of fire characteristics can be set and fine-tuned. It is only necessary to state that a certain surface can burn, at what temperature the fire point is, and how warm the surface is to start with. The triangle from the simple scene has been tilted and set to burn in the following example:

```
// light
color 0.8, 0.8, 0.8
light 150, 50, 100

// fire
burnable 1
firepoint 1000
temperature 5000

// triangle
vertex   75, -75,  0
vertex  -75, -75,  0
vertex    0, -75,  -150
triangle 0,1,2

// hack to create a big enough motionfield
vertex 0, 150, 0
sphere 3, 0
```

The result of applying the raytracer to this scene and then iterating 500 frames is shown in figure 7.4. Note the 'hack' used in the scene description, where a sphere with zero radius is used. The reason is that the calculation of the motion field automatically creates a voxel space representation of the scene. The voxel space is made as small as possible, which would only have made it one voxel high in this example if the 'hack' had not been used. So the sphere is used to let the algorithms know that the air above the triangle should be included in the computations of the motion field. Of course later versions of the raytracer could easily be modified in a way that defines the size of the voxel environment in a more elegant way.

Examples of how to use some of the other features of the parser can be found by looking at the scene descriptions in appendix C.3, which have been used to render the example

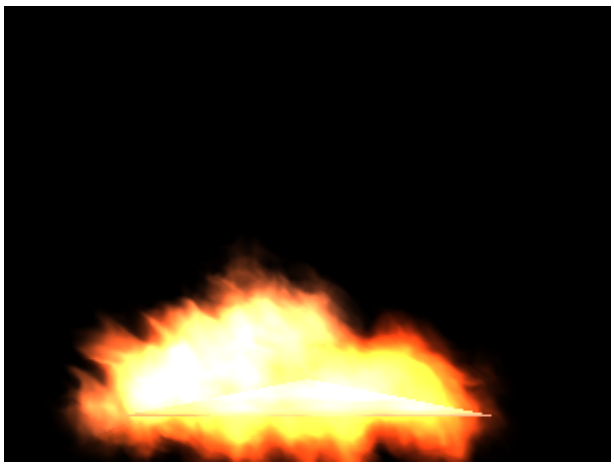


Figure 7.4: *A very simple fire scene. Notice how the fire is also visible below the burning triangle. This is due to the the volumetric properties of the fire particles being visualised by blobs*

pictures of section 8, and the complete list of parser commands can be found in appendix C.1. This section also contains an overview of some of the most commonly used command line switches used to control the level of detail.

7.3.2 Particle systems

The handling of particles is one of the areas of the raytracer where an almost unlimited number of improvements can be applied. The particle systems implemented in the raytracer at this moment are very basic and only resemble real life phenomena superficially in the way they are constructed. In this section we will describe how particle emitters have been implemented to handle the birth, colouring, movement, and death of particles, and how surfaces are made to burn.

A particle emitter is basically implemented as a position in space around which fire blobs are created. The rate of creation is controlled by a user supplied constant, but also depends on the temperature of the scene in that position. For instance no fire blobs are created if the temperature is below a user defined fire point. The particle emitter keeps a list of the fire blobs created, while they are at the same time placed in the global list of graphics primitives. In this way a particle emitter can move and in other ways modify its own fire blobs, while at the same time ensuring that the modifications are also applied to the scene being rendered. The temperature in any position of the scene, and the motion of the fire blobs are found using the animated motion field—one of the many connections that have been omitted in figure 7.1 to reduce the figure's complexity.

A particle emitter contains a list of smoke particles in addition to the list of fire particles. Whenever a fire blob enters an area of the scene where the temperature is below the fire point, it turns into a smoke blob. This effect is simply created by changing the colour and other visual characteristics of the blob. In the implemented raytracer the blob turns grey, becomes less translucent and looses its light emitting properties (obviously fire blobs emit light). Furthermore the blob is moved from the list of fire particles to the list of smoke particles. At this stage nothing has been done to make the transition from fire to smoke smooth. The radius and translucency of each smoke blob is increased as it moves through the scene, creating the effect of smoke slowly dispersing. When a smoke blob becomes so translucent that it no longer contributes any significant detail to the resulting picture, it is killed!

The colour of the fire is calculated using a very simplistic scheme: The user selects a fire colour for the particle emitter, and every fire blob is given that colour. The colour value is then scaled according to the temperature in the position of the fire blob. This has nothing to do with how real fire behaves. As shown in table 3.2 different materials burn with very distinct different colours at different temperatures. This implies that the colour of a fire should in fact be calculated by taking account of material, temperature, and oxygen saturation. The addition of these features has been postponed for future projects.

Burning surfaces

Figure 7.4 showed a very simple scene including a burning triangle. This effect was achieved by creating fire particles evenly on the surface of the triangle. Two different approaches of how to actually do this have been considered. Either a special kind of particle emitter triangle could be created, which had all the visualisation abilities of a normal triangle,

but could also create and control particles. This method would benefit from the fact that the creation of new particles could be handled in a very intelligent way, taking account of the shape of the triangle as well as texture maps or fuel maps. A downside is that this intelligent particle handling would also be very complex to implement, and would furthermore have to be implemented for each combustible kind of graphics primitive that is not made of triangles.

A far more easy approach would be to use only point sized particle emitters, and then have each combustible kind of graphics primitive define a method to spread out these emitters on the surface of the primitive. This is what we have done in our raytracer. When a triangle is made combustible a grid of particle emitters is created on the surface, the coarseness of which is defined by the user. If a fuel map has been defined on the triangle, each particle emitter is initialised differently depending on the values of the fuel map in the position of the given particle emitter. At this stage of implementation, a fuel map can only be used to describe whether a certain part of the triangle is combustible or not. Fuel maps cannot yet define, that certain areas burn more intensely or easily than others, but that should be an obvious addition in later versions. In section 8.6 results obtained by using fuel maps are shown.

When objects burn they add heat to the surroundings. This is handled in two ways in the raytracer. First of all the temperature flow of hot gas is calculated by the motion field calculations described in section 5.5.1. But these calculations only take account of existing heat, and not the heat created by the burning of the object. In the raytracer we simulate this effect by adding an amount of heat to the scene whenever (and wherever) a fire particle is created. This raises the temperature in the burning areas until a chosen maximum fire temperature is reached. When these two effects are applied simultaneously, it is possible to make a scene such as shown in figure 8.22, where a spiral shaped fuel map is used to create a fire fuse, slowly lighting up the whole fuel map.

7.3.3 Radiosity

In section 6.10.2 an algorithm for calculating radiosity in a scene was shown, along with a description of the theory involved. This algorithm has been implemented in our raytracer, taking advantage of the object oriented features of C++, and an overview of this will be given shortly. It should be noted that this is by no means the only way of implementing the algorithm, and it is probably not the fastest. The design has been chosen for the sole purpose of helping the programmers (us) to avoid confusion when calculating radiosity between different types of graphics primitives.

The algorithm distinguishes between blobs and surfaces (the only surfaces implemented are triangles) both in the way the form factors are calculated, and in the amount of samples used to approximate the light properties on the surface. A blob is approximated by a disc, having a normal pointing directly toward the target, while a triangle is approximated by a number of small discs, each having the same normal as the triangle (See figure 6.18 on page 99). This means that different calculations must be used when calculating the radiosity from a blob and a triangle. When calculating the radiosity *recieved* by a graphics primitive, the same differences apply: A blob only recieves light in one point, while a tringle receives light in a number of sample points. The different equations were shown in section 6.10.2.

The idea of implementing a central radiosity function that distinguishes between blobs and triangles, and then chooses the correct formulas depending on what type of primitive

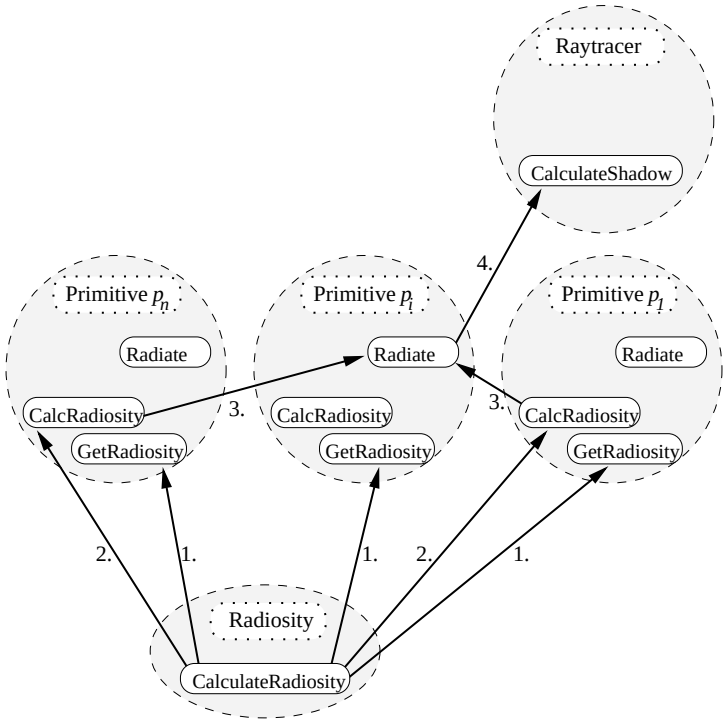


Figure 7.5: *An overview of how the radiosity calculations are applied by the graphics primitives. The primitive p_i radiates on the other primitives. Only the relevant methods of the primitives have been shown.*

lights up what other type of primitive does not sound feasible. Instead, by using an object oriented approach, we let the graphics primitives control how they radiate and receive light. Each primitive just needs to know how to radiate light to a position in space, and how to receive light from some (arbitrary) source. Take, for instance, a triangle. As mentioned, the light falling on a triangle is sampled in a grid. When the triangle is asked to calculate the amount of light it radiates to any point, each sample point is used as a disc approximation, and the radiosity from these discs to the point is summarised. On the other hand, if the triangle is asked to calculate the amount of radiosity received from any other primitive, the triangle just runs through the sample points, and asks the other primitive to calculate the radiosity to each of these points. In figure 7.5 this interaction between graphics primitives is shown (compare this figure with figure 7.1.)

The figure shows the steps performed when calculating radiosity and the interactions between the graphics primitives at each step. These steps follow the general radiosity algorithm described on page 94. The numbers at the arrows represents the following actions:

1. The primitive that radiates the most light is found (p_i in the example figure). This is done by calling the `GetRadiosity()` method for each primitive.

2. All other primitives should now receive radiation from p_i, so the `CalcRadiosity()` method is invoked for each of these in turn. p_i is supplied as a parameter, so the primitives know which primitive they receive light from.

3. In the `CalcRadiosity()` method the primitives know which points must receive light from p_i. For each of these points they call the `Radiate()` function in p_i, which calculates form factors and radiosity from p_i to the point. When calculating form factors, it is necessary to determine if any light is occluded between the point and the primitive, so step 4 is executed to determine if any objects lie in the line from the point to the primitive.

4. p_i calls `CalculateShadow()`, which calculates the fraction of light being occluded between two points. This is done by examining intersections between a ray and all primitives, just as when doing normal shadow calculations in the raytracing algorithm (as described next).

7.3.4 Raytracing algorithms

We have implemented a basic standard type raytracer as described in [Watkins et al., 1992], but with the addition of a few not quite trivial features. The calculation of radiosity has been treated in section 7.3.3, but features such as translucent primitives and backwards warping of blobs need a description.

As long as no graphics primitives are translucent it is only necessary to find the intersection (for each ray) which is closest to the camera. With translucent primitives it is necessary to find a sorted list of intersections instead, as the ray might continue through the primitives encountered. (This assumes that the primitives do not *bend* the ray). To complicate matters, translucent spheres might intersect the ray twice, perhaps with other primitives intersecting inbetween. The simple way of handling this is to let the sphere appear twice in the list of intersections—once when the ray enters the sphere, and once when it leaves the sphere. The raytracing is then performed as usual in the two intersections.

It becomes really interesting when blobs are involved, as the intersections between a blob and a ray defines the points between which the blob must be sampled (see section 6.9.4). Simply splitting a blob up in a start intersection and an end intersection and treating each point independently will not be of any use, as the blob visualisation algorithm must work on intersection intervals—not intersection points.

In section 6.9.4 different ways of sampling overlapping blobs were proposed. Here it was argued that when two blobs of similar appearance overlapped, it would probably be sufficient to fully sample and visualise each blob in turn. Obviously blobs and solid surfaces are not of similar appearance, so the arguments do not apply in the case where a group of overlapping blobs is bisected by a triangle. In our raytracer the list of intersections is used to find intervals that only contain blob intersections (the solid graphic primitives intersection points are used as interval start and end points). Inside these intervals the blobs are treated as described in section 6.9.4, apart from the fact that a bit of housekeeping is needed whenever a blob starts in one interval and ends in another.

Look again at figure 7.1 (or figure D.1 in appendix D.4 for more details). The function `Trace()` in the raytracer first builds the sorted list of intersections using the `Intersect()` method in each primitive. Now the `GetIntersectionColor()` method is invoked for the graphics primitives in the list. This method calculates the translucency as well as the visual features that are specific for each type of graphics primitives. For instance the backwards warping of blobs (section 6.9.3) is handled here, using the motion field and a precalculated second level turbulent wind field. For triangles, texture mapping and radiosity light map

interpolations are applied here.

The calculation of shadows is pretty straightforward. Graphics primitives intersecting a light ray can be found with the primitive's `Intersect()` method, and the amount of light being occluded can be found by using a specialised call to `GetIntersectionColor()`, that only calculates translucency and not colouring of the primitive.

7.4 Runtime Complexity of the raytracer

This section contains an analysis of the complexity of the different parts of the raytracer. With these considerations it should be possible to determine which algorithms are the most time consuming, and what effect any conceivable future improvements would have on these algorithms.

The rendering of an image is subdivided in a number of independent steps. The initialisation, the calculation of motion field, the calculation of radiosity and finally the raytracing. In the following, the complexity of each of these steps will be described. When sorting lists of graphics primitives we use a sorting algorithm of complexity $\mathcal{O}(n^2)$.

In the following n will be the number of graphics primitives, l the number of light sources, w the backwards warp steps, and k the blob sample steps.

7.4.1 Initialisation and calculation of motion field

In this part of the rendering process the main amount of time is used calculating the animated motion field. These calculations are performed iteratively on a voxel representation of the scene, and the complexity is linear in the number of voxels (or cubic in the 'resolution of the voxel space'. As this amount is chosen by the user it can easily become huge, but due to the linear complexity this should not be a problem, unless so large an amount is chosen that the computer's amount of memory does not suffice.

7.4.2 Radiosity calculation

Section 7.3.3 shows how radiosity is calculated in our raytracer implementation. First the amount of light received on each graphics primitive from each light source is calculated. This is done by taking the light rays from each light source (l) to each primitive (n) and then intersecting them with each primitive (n). If the intersecting primitives are blobs, they must be sampled and backwarped (kw). All in all we get $\mathcal{O}(n^2lkw)$

After this the radiosity is found by iterating the light in the scene until the results are satisfactory. A maximum number of iterations used can be set by the user. Let us say we will at most apply i iterations. For each iteration the graphics primitives are first sorted according to the amount of reflected light (n^2). *Then* all the primitives of the list are set to radiate light on all other primitives (n^2). If shadow calculations are not enabled $\mathcal{O}(n^2i)$ is then the resulting complexity. Otherwise, the light ray between each pair of graphics primitives must be tested for intersection with all primitives, to calculate the amount of light being occluded along the ray (n). If the intersecting primitives are blobs, they must be sampled and backwards warped (kw). All in all we find the complexity for the radiosity algorithm to be $\mathcal{O}(n^3kwi)$ when shadow calculation is included.

7.4.3 Raytracing

When raytracing the scene, the following steps are performed for each pixel in the output image:

A ray is shot from the centre of projection through the pixel on the image plane and tested for intersection with all graphics primitives. These are then inserted in a sorted list according to the intersection distance (n^2). *Then* the list is used to examine the primitives one by one. Each blob primitive is sampled and backwarped (nkw, if the simple blob sample algorithm is used. See section 6.9.4). Each sample point must then receive light from each light source. If the shadow calculation is included, this means that a light ray from each light source to the sample points must be tested for intersection with all graphics primitives. If the primitives are blobs, they must be sampled and backwarped ($nkwl$). So all in all the complexity of the raytracing algorithm is $\mathcal{O}(n^2) + \mathcal{O}(n^2k^2w^2l) = \mathcal{O}(n^2k^2w^2l)$.

7.4.4 Possible Improvements

As seen above, the complexity of a complete raytracing including radiosity and shadow calculations is: $\mathcal{O}(n^3kwi) + \mathcal{O}(n^2k^2w^2l)$. The k, w, l and i parameters are usually kept very small. In almost all examples of chapter 8, k and w have been set to 5, i is usually set to 2 and there is rarely any point in letting i exceed 10, and l only exceeds 10 for *very* complex scenes. As the number of primitives easily reaches several thousand when fire and smoke blobs are created, it is probably safe to ignore the k, w, l, and i factors when discussing the overall complexity. We therefore find the complexity of the raytracer to be: $\mathcal{O}(n^3) + \mathcal{O}(n^2) = \mathcal{O}(n^3)$.

The above considerations show, that an improvement of our $\mathcal{O}(n^2)$ sort function will not change the overall complexity of the raytracer, as the radiosity algorithm is $\mathcal{O}(n^3)$ independent of the sort function. The same goes for the $\mathcal{O}(n^2)$ complexity of the raytracing algorithm. So by these arguments the choice of a simple sorting algorithm does not *seriously* slow down the raytracer. When shadow calculations are disabled the complexity of the radiosity calculations drops to $\mathcal{O}(n^2)$ and now the complexity of the raytracing algorithm actually depends on the sorting algorithm and can be improved to $\mathcal{O}(n \log n)$

The above considerations makes it clear that the program would benefit the most from an improved radiosity algorithm. Undoubtedly faster radiosity algorithms have been created, but as we find that radiosity is a rather peripheral area when discussing modelling, animation and visualisation of fire, we have chosen not to pursue this subject any further. Furthermore we have never intended our raytracer implementation to be fast. The point has always been to be able to show good examples of what can be done.

After working with the implementation and the algorithms we have found a few areas that could relatively easy be sped up—at least in the general case. None of these speed-ups change the complexity in the worst case scenarios, but they should work well even if the radiosity and raytracing algorithms are changed.

Discarding precision in the radiosity calculations

First of all, the radiosity calculations could be sped up by discarding the radiosity between objects that are too far apart to reflect any noticeable amount of light on each other. As the light contribution from one object to another is reduced by the square of the distance (see section 6.10.2), it should be possible to discard a lot of calculations in big scenes.

Alternately it should be possible to keep control of which surfaces can never be hit by direct light from which other surfaces, for instance by using data structures such as binary space partition trees (see [Foley et al., 1990]).

Hierarchical grouping of blobs

During raytracing, a lot of time is consumed by repeatedly testing for intersections between some ray and all blobs in the scene. As blobs are usually small but numerous and tend to lie fairly close, a hierarchically grouping of the blobs should reduce the number of intersection tests needed. Seefigure 7.6.

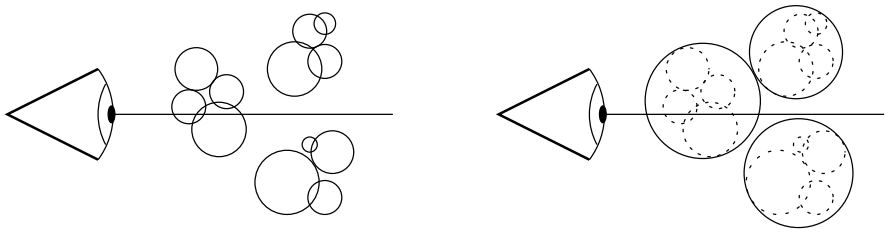


Figure 7.6: *Reducing the number of blob intersection tests in the general case by grouping the blobs hierarchically. In the example on the right the upper and lower group of blobs are discarded leaving only four blobs to be checked for intersection. The speedup is 5 intersection tests.*

The figure shows a setup with a ray being shot through a cluster of blobs. In the picture on the right, the blobs have been divided into three subgroups. When testing the ray for intersections, 12 tests are needed in the left setup, whereas only seven tests are needed on the right, as the first three tests discard the eight blobs in the two groups that do not intersect the ray. The subdivision can be carried on further, ending up with *one* group consisting of several subgroups, each consisting of several subgroups and so forth. This way it will be possible to discard *all* blobs (which might be several thousand) using only one test if the ray does not intersect the big group of blobs. In the worst case—though—all groups, subgroups and blobs intersect any given ray, which actually slows the raytracer down. For instance, if all blobs in figure 7.6 intersected the ray, the amount of tests needed would rise from 12 (without blob grouping) to 15. In the general case a hierarchical grouping of the blobs results in good speed-ups, as demonstrated by [Stam, 1995].

Bounding boxes

Another possible speed-up along the same lines would be to use bounding boxes for all graphics primitives in the scene. This could be used for quick tests as to whether a primitive *might* intersect a ray or not. For instance, it is possible once and for all to calculate which pixels each primitive might occupy. This way it will not be necessary to test for intersections with primitives in the areas of the resulting image where these primitives do not contribute.

Chapter 8

Results

"To them the truth would be literally nothing but the shadows of the images"

Plato, "The Republic"

This section contains test data produced by our raytracer. The examples are chosen to fulfil three different purposes:

Expectation To demonstrate that the chosen modelling, animation and visualisation algorithms produce the desired results.

Illustration To illustrate the effect of applying the methods discussed in the theory to a scene. This should give a good visual understanding of the usefulness of these methods. It should also show which parts of the theory that are good, and which parts that need improvement.

Comparison To compare visual quality with rendering time. Whenever more detail is added to the raytracing algorithms, image quality should hopefully improve. But the rendering times might skyrocket at the same time. It is important to find a tradeoff that results in sufficiently good images while still being fast enough to get the job done in time. All tests have been performed using a HP9000/C160 with 256 MByte RAM running HP-UX 10.

The experiments have been divided into these parts: Radiosity (8.1), blob visualisation (8.2), shadowing algorithms (8.3), motion field (8.4), second level turbulence (8.5), animation (8.6), image quality vs. rendering times (8.7), and simulation of scattering (8.8). Scene files and program parameters can be found in appendix C.3. Each section will start with a minor review of the theory involved. These reviews show why the chosen examples are interesting.

8.1 Radiosity

The radiosity algorithm implements the fact that all surfaces emit or reflect light to some extent. For instance, if you were to shine a bright light on a white piece of paper, the paper would 'brighten up' and in turn light up its surroundings. In this way the light can be reflected to reach surfaces that would otherwise lie in total darkness. The radiosity algorithm uses a value called the *reflectivity* of the surface. This value states how big a fraction of the incoming light (or energy) that is reflected to other surfaces. The radiosity of a surface is calculated by sampling the surface on a grid, and then approximating the radiosity values in the sample points creating a *light map* on the surface. The value radiositymap defines the resolution of this sampling in the scene input file. When visualising the surfaces, the light map is interpolated, creating a smooth change of lighting. Radiosity is also used to create light sources which are not point-sized. The algorithm calculates the radiosity of all surfaces by iteration. In each iteration all surfaces emit light to all other surfaces. To reduce the number of iterations necessary, the sequence of surfaces is chosen so the ones that emit the most light are evaluated first. A detailed description of the radiosity algorithm is found in section 6.10.2.

8.1.1 Simple radiosity between two surfaces

Let us start by testing a very simple scene. Two surfaces are situated close to each other. A light source shines on part of one surface, but only hits the back side of the other surface. This is the setup of figure 8.1, where ambient light has been added, so we can see the top surface.

Figure 8.1: *The setup for the simple radiosity test. Any of the scene descriptions* `testrad1b.tsr`, `testrad1c.tsr`, *or* `testrad1d.tsr`, *will result in this picture if rendered without radiosity.*

The radiosity calculations are now applied to the scene to see what effect the reflectivity has on the result. The reflectivity is set to 0.5, 1.0, and 1.5 respectively. Note that a reflectivity of 1.5 means that actually *more* light is reflected than is received. This is obviously impossible and could cause the scene to light itself up to an infinite light amount if the surfaces were close enough. In this example—however—it just makes the effect of raising the reflectivity more evident. Figure 8.2 shows the different results obtained.

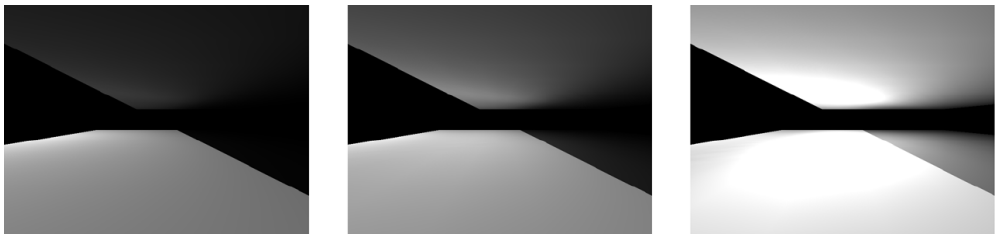


Figure 8.2: *Radiosity between two close surfaces. In the leftmost picture, the reflectivity is set to* 0.5. *In the middle picture the reflectivity is* 1.0, *and in the rightmost picture it is* 1.5. *The scene descriptions are* `testrad1b.tsr`, `testrad1c.tsr`, *and* `testrad1d.tsr`.

8.1.2 Radiosity in a complex room

In this section we will examine how the radiosity algorithm facilitates the spread of light in a more complex scene. The scene shows a room with a big box standing against the back wall. Behind the box is a powerful light source. The scene is shown in figure 8.3, where no radiosity has been calculated.

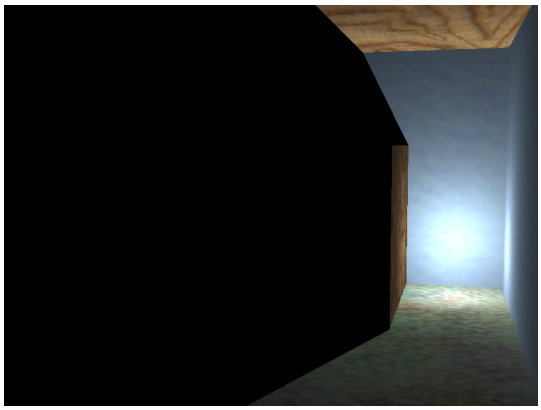


Figure 8.3: *The setup for the complex radiosity test.*

The goal of the experiment is to see which effect the number of radiosity iterations has on the the lighting. It is clear that for each step of the algorithm, more light should be reflected to the dark regions of the room. It should also be clear that every iteration of the algorithm adds light to the scene, as one extra iteration corresponds to light being reflected one more time.[1] Therefore a setup is included, where the algorithm is iterated 'many' times, to see that the result converges. If not, either the program is buggy, or very dark sunglasses would be required in our every day life.

An interesting aspect of this test is to examine, how coarse the light map can be, and how this corresponds to rendering times. To do this, the same scene is rendered with different resolution of the light map/radiosity map. Figure 8.4 shows the two scenes for 1, 2, 4, and 10 radiosity iterations. Table 8.1 shows the rendering times involved. The pictures have been rendered in 800×600 pixels.

Radiosity Iterations	**1**	**2**	**4**	**10**
6×6 **radiosity map**	11.5	20.9	40.3	97.7
11×11 **radiosity map**	83.4	164.8	325.9	817.7

Table 8.1: *Timing the radiosity calculation. This table shows the time (in seconds) used for calculating the radiosity. To render the final images of figure 8.4 a further* 43.2 *seconds are needed.*

As table 8.1 shows the increase of radiosity map resolution results in a slowdown by a factor 8. In section 7.4 we argued that the radiosity calculations have a complexity of $\mathcal{O}(n^3)$ when calculating shadows and $\mathcal{O}(n^2)$ when not. Increasing the radiosity map resolution increases the number of sample points on each surface, which affects the general radiosity calculations but does not affect the calculations of shadows (the number of sample points is irrelevant when calculating whether or not a surface is intersected by a light ray). Thus the rendering time should be influenced by the square of the increase in radiosity map

[1] Actually the algorithm is better than that, as the sequence of radiating surfaces is chosen in a way that lets the light from the brightest surface be reflected to all surfaces. After that, it can be reflected from the second brightest surface to all other surfaces and so forth. This permits the light from the brightest surface to be reflected n times in one iteration, where n is the number of surfaces in the scene

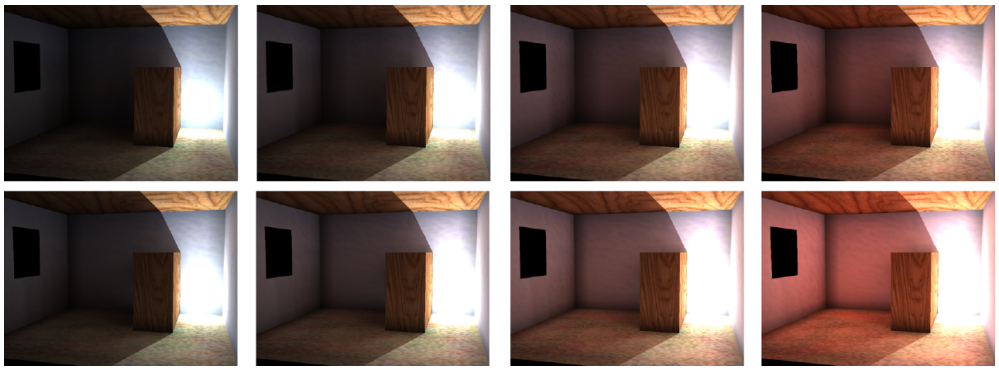


Figure 8.4: *The radiosity algorithm performed on a more complex scene containing a point light source. The top row of pictures uses a* 6×6 *radiosity map for the radiosity algorithm, whereas the bottom row of pictures uses a* 11×11 *radiosity map. The number of radiosity iterations are (from left to right): 1, 2, 4, and 10. The scene descriptions are* `testrad2.tsr` *and* `testrad3.tsr`. †

resolution.

Using a 6×6 radiosity map on a triangle results (in our implementation) in the creation of 21 sample points[2] on the surface of the triangle, whereas an 11×11 radiosity map results in 66. The expected increase in calculation time would then be of a factor $(\frac{66}{21})^2 \approx 9.88$. The error in this estimate is probably caused by the overhead of time used for the shadow calculations.

As the pictures of figure 8.4 clearly show there is a sharp shadow being cast from the point light source in the scene. This shadow only exists because our algorithm still uses the point light sources when rendering the scene, even though radiosity calculation is involved. If 'only' radiosity is used, a sharp shadow is difficult to create, as the radiosity map is interpolated to find the lighting in a given point. As a result all shadows are smoothed, depending on the resolution of the radiosity map. In figure 8.5 we show how this affects the scene. The point light source has been substituted by a very small sphere with a very high amount of self illumination. As this is a graphics primitive and not a light source, the raytracer no longer has any light sources to use for casting sharp shadows. All shadows are therefore a result of interpolation in the radiosity map. Notice that this can entail that shadows do not "fit" properly to the side of the box. This problem is obviously bigger the smaller the radiosity map is.

8.2 Blob visualisation

Blobs are created for two purposes: To visualise *particles* (see sections 5.2 and 8.4) and to create an illusion of 'dusty' or light-emitting gas. In this section we will examine the properties that make blobs usable for simulating turbulent smoke (or fire) effects.

A blob is basically a position in space surrounded by a density field. This density field describes how 'solid' (or *translucent*—depending on how you choose to view it) the blob is in a given position. When raytracing through a blob the total translucency of the blob would have to be found by integrating the blob translucencies along the ray. The algorithm used

[2]Six sample points from the first row in the map, five from the next, four from the next, and so forth.

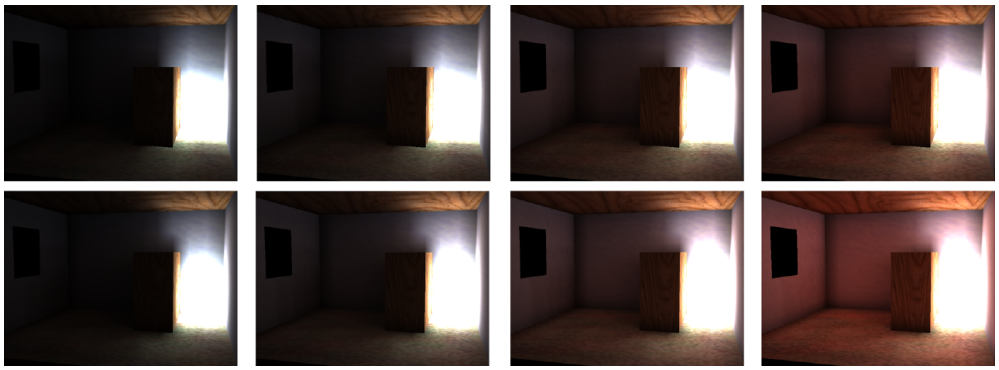


Figure 8.5: *The radiosity algorithm performed on a more complex scene containing a self illuminating sphere as a light source. The pictures illustrate the 'soft shadows' that result from interpolating the radiosity map. The top row pictures use a* 6×6 *radiosity map, and the bottom row pictures use a* 11×11 *map. The scene descriptions are* `testrad4.tsr` *and* `testrad5.tsr`. †

to visualise a blob approximates this integral by dividing the blob in a number of *sample steps*. At each step the translucency is approximated and thus the total translucency along the ray is found. Obviously the number of sample steps greatly influences the correctness of the result. When two or more blobs overlap, it is not obvious how the sample points should be chosen. We have created two different algorithms for sampling the blobs in the scene. One is fast (few sample points) and one is precise. More detail can be found in section 6.9.4.

The thing that makes blobs so useful for creating smoke-like effects, is that it is possible to *warp* a blob according to a given motion or turbulence field. This is done by taking the sample points of the warped blob, and then move them backwards through the field along with the centre of the blob. The resulting values are the used to find the blob translucency in the sample points. The number of steps that the sample points are moved is called the number of *backward warp steps* or *backwarp steps* for short. This backwarping can be thought of as a way of using few blobs to simulate the results you would get if many blobs were used. A more detailed description of blob backward warping is found in section 6.9.

8.2.1 Blob warping and sample steps

When simulating smoke using backwards warping of blobs, turbulence is very important. The amount of backwards warp steps makes the turbulence more noticeable, while the number of sample steps makes the result more precise. The question is if precision is really that important. As it is, the turbulence is just a kind of structured noise, and small errors in this noise probably have no effect in the subjective judgement of the result. In this experiment the number of sample steps are modified to test this assumption . The values used are: 1, 3, 5, 7, and 9. It is not certain, that many backwarp steps give 'better' results. It might be possible to over-do the turbulence. This depends on the desired kind of special effect. To give some insight in these matters, the experiment also changes the number of backwarp steps, using these values: 0, 4, 8, 12, and 16. Figure 8.6 shows the pictures that result from the different combinations of sample and backwarp steps to a single blob in a turbulent field. The pictures have been rendered with no shadows or highlights in 512×512 resolution, and the rendering times can be found in table 8.2.

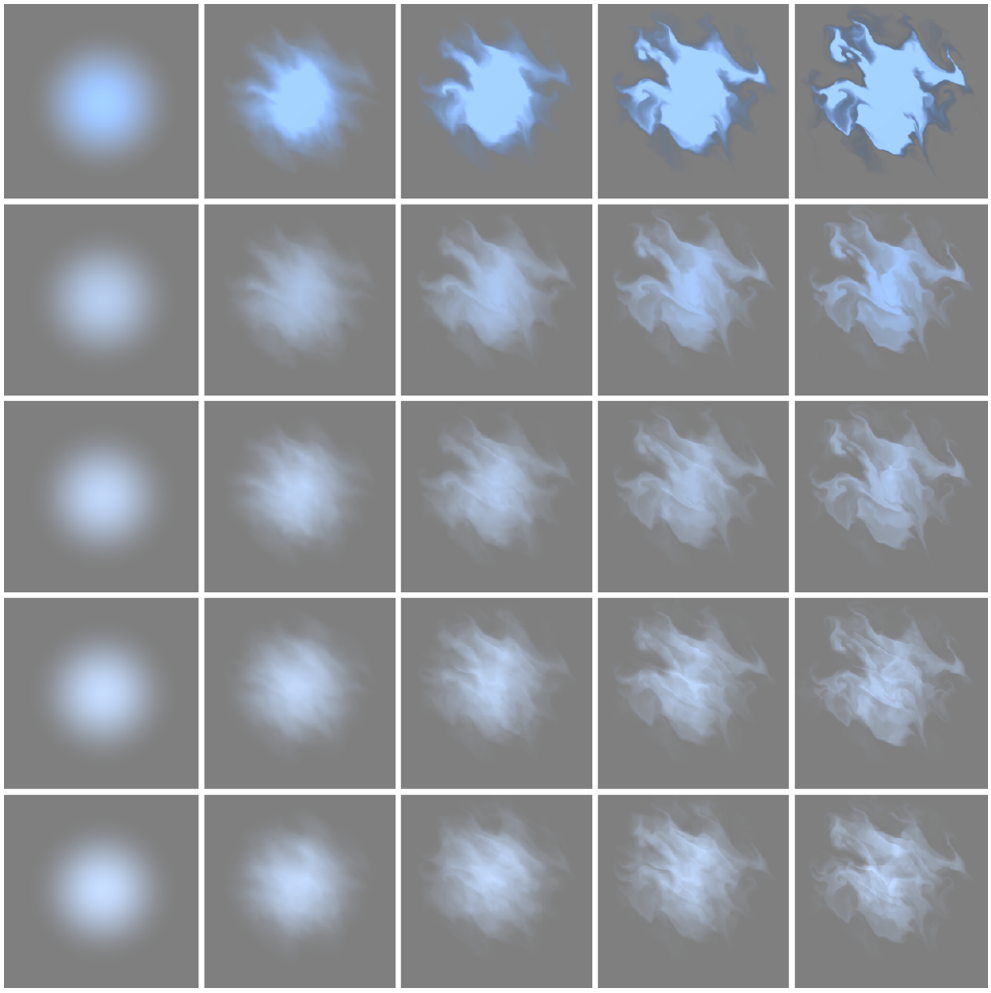


Figure 8.6: *This figure shows the importance of the number of blob backwards warp steps in relation to the number of blob sample steps. The number of backwards warp steps is (each column, from left to right): 0, 4, 8, 12, and 16. The number of blob sample steps is (each row, from top to bottom): 1, 3, 5, 7, and 9. The scene description is found in* `testwarp1.tsr`. †

sample \ backwarp	**0**	**4**	**8**	**12**	**16**
1	4.5	35.5	70.1	103.5	136.1
3	5.6	93.1	203.0	304.8	403.3
5	6.7	152.3	334.7	502.7	666.1
7	7.8	211.9	467.4	698.3	928.1
9	8.9	269.3	593.2	896.1	1209.3

Table 8.2: *Rendering times in seconds of blob sampling and backwarping. The numbers clearly show that the rendering time for a single blob using no shadowing is $\mathcal{O}(kw)$ where k is the number of sample steps and w is the number of backwarp steps.*

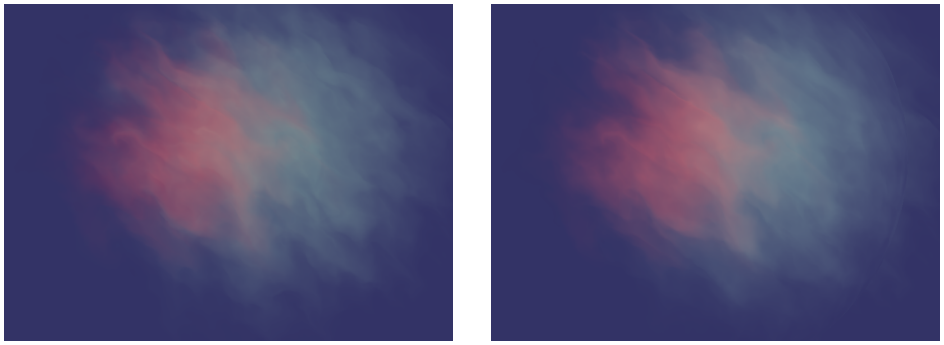


Figure 8.7: *This figure shows the differences obtained by using the two different blob sample algorithms to a scene containing two overlapping blobs (`testwarp2.tsr`). The leftmost picture is obtained using the simple blob sample algorithm, while the rightmost picture uses the more precise algorithm. The algorithms are described in section 6.9.4.* †

8.2.2 Blob sample algorithm

As mentioned in section 6.9.4, two algorithms for calculating the blob translucency and colour along a ray are implemented. One should be fast but imprecise, while the other should take a great deal more time, but also deliver better results. To test this, a scene consisting of two overlapping blobs differing in translucency and colour has been created. The pictures in figure 8.7 have been created by applying the different blob sample algorithms to the scene.

The figure shows, what at first glance looks like two identical pictures. Closer examination reveals, that any increase in image quality obtained by the more precise algorithm is restrained to almost un-noticeable details, and it is doubtful if this quality increase is worth the extra rendering time. The time consumption of the two algorithms are compared in table 8.3. In this test, we compare the rendering times for a scene with two overlapping blobs and for a scene with 20 overlapping blobs. The resulting images are not shown, but they have been rendered with shadows disabled and in a 100×100 resolution. As the simple blob rendering algorithm is $\mathcal{O}(kn)$ (see section 6.9.4) we would expect an increase of rendering time by a factor 10 when going from 2 to 20 overlapping blobs. The more precise algorithm—on the other hand—has a complexity of $\mathcal{O}(kn^2)$, so in this case a factor 100 is expected. The results illustrate how this is in fact the case.

Sample Algorithm	Overlapping blobs	Time
Simple	2	16.6
Simple	20	165.9
Precise	2	39.5
Precise	20	4526.9

Table 8.3: *Rendering times depending on blob sample algorithm. The time is measured in seconds.*

8.3 Shadowing algorithms

In raytracers shadowing algorithms generally work by calculating the amount of light that reaches a given point of a graphics primitive from the light sources in the scene. This is done by shooting rays from the lights to the point. If a light ray intersects an object the light will be stopped unless the object is translucent, in which case the light intensity will be reduced but the light ray will continue.

Calculating shadows takes a lot of time, especially when using blobs. We must remember, that to find the 'shape' of a blob, it is necessary to sample it and then backwarp the samples. So if a blob is sampled in n steps, backwarped w steps, and we have l light sources (l is very big when calculating shadows during the application of the radiosity algorithm, as each point in a graphic primitive's radiosity map is treated as a light source.), each blob will add wln backward warpings to the algorithm. As a backward warp consists of three four-dimensional interpolations it is not fast, so the time consumption is considerable.

To deal with this, the shadowing algorithm in the raytracer has four levels of shadowing detail. The fast 'no shadows' option, where all shadow calculations are completely disabled. The more time-consuming 'simple shadows' option, where shadows are calculated for all objects except blobs. The even more time-consuming 'simple blob shadows' option, where shadows are enabled for all objects, but the blob shadows are simplified. This is done by ignoring backwards warping and only sampling the blobs in two steps when calculating shadows[3]. Lastly we have the 'full shadow' option, where all shadows are calculated as accurately as possible.

In this section we examine the effect of these shadowing options. In figure 8.8 a scene is shown, where the light from a point light source is intersected by a blob, before reaching a surface. This surface partly covers a second surface lying behind it.

The figure shows how the different shadowing algorithms affect the outcome. In the top left picture all shadowing has been disabled, so both surfaces and the blob are fully lightened. The top right picture is created using the 'simple shadows' option. It shows that the light passes unhindered through the blob, but the first surface casts a shadow on the second surface. In the bottom left picture the 'simple blob shadows' option has been used. Now the blob casts a shadow on the first surface, but the shadow does not resemble the blob in any way. It is clear that this is the shadow of an unwarped blob. Finally all shadowing is enabled in the bottom right picture. Here the blob casts a correct shadow.

The effect of increasing the shadowing algorithm's complexity can be viewed in table 8.4, where the rendering times of the pictures in figure 8.8 are listed. The pictures have all been rendered in 800×600 pixels, and with 5 blob sample steps and 5 blob backwarping steps.

As the results clearly show, it is possible to save a great deal of time by raytracing scenes with a simpler shadowing option. In some cases the loss of quality will even be relatively small. For instance, if the scene contains a lot of fire blobs and few smoke blobs (that is—the 'amount' of shadow cast by blobs is small) then it will probably be safe to use the simple shadows option. If the scene contains a lot of smoke, it will make a noticeable difference if the smoke blobs do not cast shadows on each other. In this case, the simple blob shadow option can be used. As we have just seen, the shadows cast by blobs become

[3]It has been tried to sample the blob in one step, but this resulted in too dark shadows, as the blob translucency along a ray was approximated by the value closest to the centre of the blob; the least translucent place.

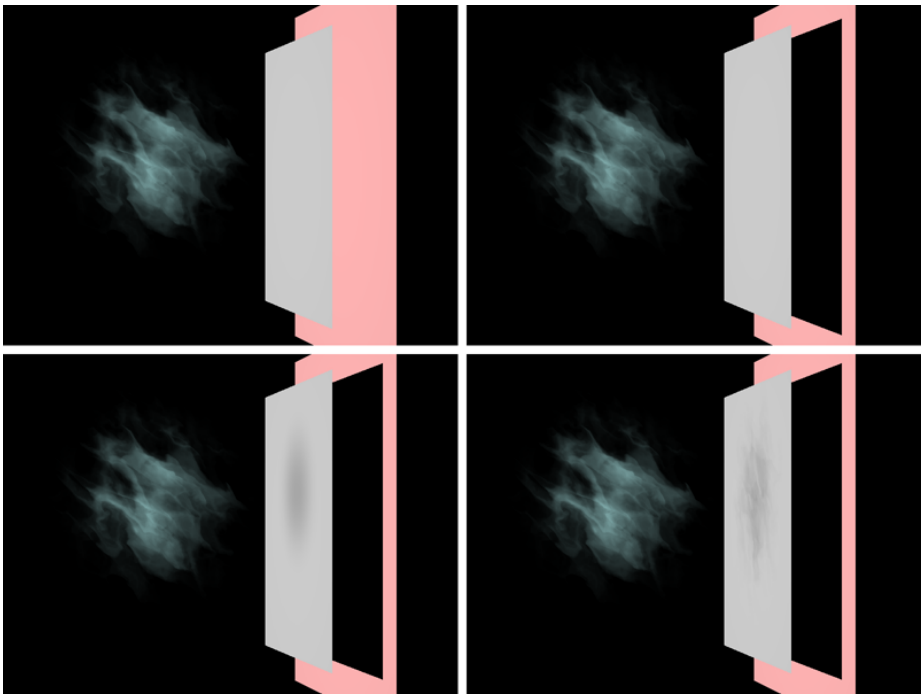


Figure 8.8: *This figure shows the differences obtained by using four different levels of shadowing. The top left picture is created using 'no shadows', the top right uses 'simple shadows', the bottom left uses 'simple blob shadows', and the bottom right uses all shadows. The scene description can be found in* `testshadow1.tsr`. †

Shadowing option	Rendering Time
-s (No Shadows)	163.1
-ss (Simple shadows)	165.6
-sbs (Simple blob shadows)	167.5
Full shadow calculation	444.8

Table 8.4: *Rendering times depending on shadowing options. The time is measured in seconds.*

very regular and smooth with this option. This is not a big problem, however, unless point light sources are used. As seen in section 8.1.2, the lighting from the radiosity algorithm becomes very smooth itself, so if the scene is raytraced using only radiosity light, the simplified blob shadows will be very hard to notice. If a still picture of smoke is wanted, with point light sources used for illumination, the full shadow option must be used.

We will end this section by showing what happens when it is not the blob that casts a shadow on a surface, but a surface that casts a shadow onto a blob. For this purpose, a scene has been created in which two light sources—a red and a blue—lights up a blob. The blue light source lights up the blob from the left, except that the blob is partly obscured by a surface. The red light source lights up the blob from the lower right of the scene. As the results in figure 8.9 show, the blob is coloured by the light, and in the configurations where shadows are enabled, it is clear how the blob becomes two-coloured by the shadow

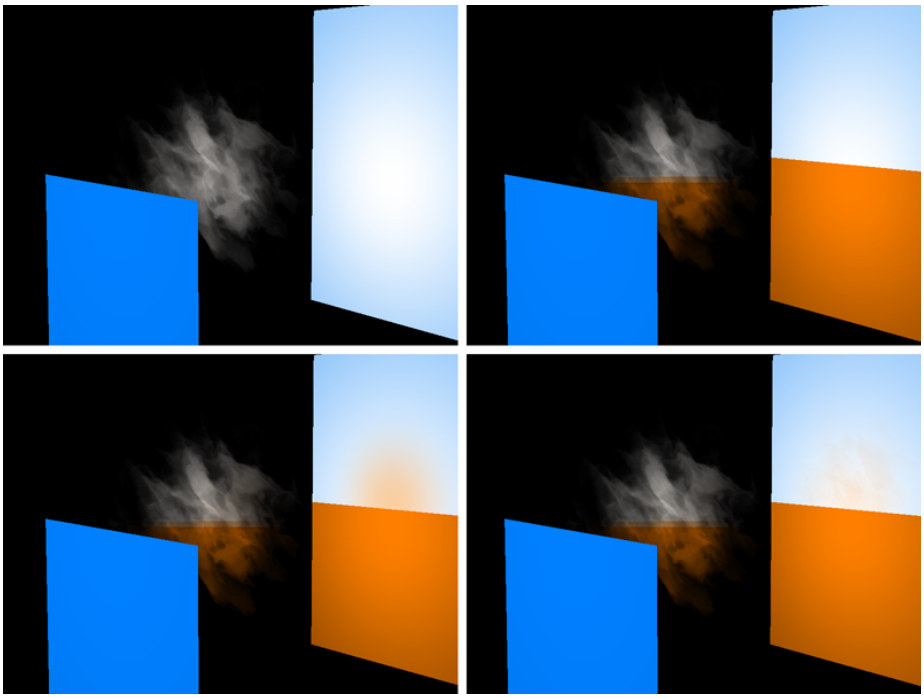


Figure 8.9: *This figure also shows the differences obtained by using four different levels of shadowing. Here we focus on a surface casting shadow on a blob. The top left picture is created using 'no shadows', the top right uses 'simple shadows', the bottom left uses 'simple blob shadows', and the bottom right uses all shadows. The scene description can be found in* `testshadow2.tsr`*. The figure is repeated in colour in appendix B.* †

from the leftmost surface. When 'full shadows' or 'simple blob shadows' are enabled the blob casts a shadow on the second surface. The shadow is red, because it is the blue light that is obscured. The figure is shown in colour in appendix B.

8.4 Motion field

In the raytracer, smoke and fire effects are created by moving and warping blobs in the scene. The turbulent details of the warping come from using a turbulent wind field. This has been demonstrated in section 8.2. The turbulent wind field is a precalculated 4 dimensional noise array that has been created in a way that makes it very useful for adding a turbulent look to a blob (see section 5.4). It is not—however—as useful for moving blobs, as it quickly becomes very clear, that the resulting motion in no way corresponds to the effects caused by solid objects in the scene. For instance, a naïve algorithm might allow smoke to drift through the ceiling. Even if this was somehow taken care of, the amount of turbulence would be the same everywhere regardless of walls and obstacles, which is not the way turbulence works in real life.

To overcome these problems, an animated motion field is created as shown in section 5.5.1. This field subdivides the scene into small cubic entities called *voxels*, and uses local temperature and motion vectors (as well as information about whether a given voxel is part of a solid object) to calculate an approximation to the physically correct motion field of

the given scene. The amount of turbulent detail captured by the field depends on the *voxel resolution*—the size of each voxel, where a high voxel resolution (many small voxels) better captures the finer turbulence. If an infinite voxel resolution could be used, there would be no need for the addition of *second level turbulence*—the turbulence from the precalculated turbulent wind field described above. In this light, the second level turbulence can be viewed as a way of greatly reducing the animation time, without loosing too much detail.

The motion field is not static. For each frame of animation (*animation step*) the motion field is updated according to the most recently calculated values. In this iterative way, motion in one part of the scene will disperse out to the whole scene over time. This is used when creating an animation of fire. The fire is hot, and creates an upward motion. This warm rising gas then mixes with the colder air above, and a turbulent motion is created.

The visualisation of fire or smoke in the animated motion field is done by using *particle emitters*. A particle emitter is an entity that creates and animates blobs according to some simple rules. The blobs are created according to the temperature in area of the particle emitter, and they are moved using the animated motion field. To create the appearance of a burning surface, particle emitters are placed at regular intervals on the surface. Increasing the number of emitters on a surface (the *emitter resolution*) adds detail to the resulting flame, but also increases rendering time. This is the tradeoff between many small blobs or fewer big blobs. The backwards warping of blobs helps to raise the quality of pictures using few big blobs.

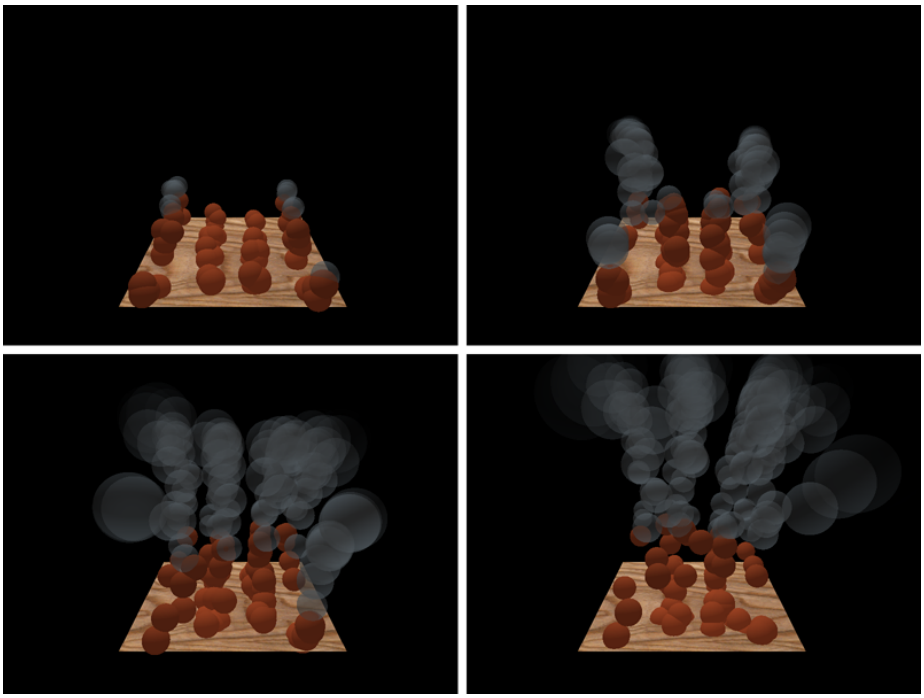


Figure 8.10: *Four pictures showing the effect of calculating an animated motion field in a simple* $15 \times 9 \times 15$ *voxel environment. The pictures have been raytraced after 200, 400, 600, and 800 iterations of motion field and blobs. The scene descriptions can be found in* `testmotion1.tsr`. †

8.4.1 Simple motion field

We start the tests by examining the effects of the voxel resolution. To do this, a simple scene has been created containing a burning square. When the test starts, the motion field is zero everywhere, as the air in the scene is completely still. The test shows, how the motion field changes as it is iterated. The particle emitters are set to emit spheres instead of blobs, to make the position of each particle clear, and to speed up the raytracing process. It should be noted that the spheres are smaller than the blobs would be, so that each sphere can be distinguished from its neighbours. Had the scene been rendered with blobs, they would have been big enough to overlap. In figure 8.10 results are shown for a voxel resolution of $15 \times 9 \times 15$. The motion field and the blobs have been iterated 200 times between each picture, and the evolution of the motion field can easily be seen. Notice how the spheres are pushed towards the centre of the 'flame'. This is due to the hot gas rising, causing the pressure to drop below the gas, which in turn causes the surrounding gas to be 'sucked in' from the sides. On the other hand, the gas at the top is forced to the sides when it encounters the cooler air above.

The results look plausible, even though the voxel resolution is very low. This has a lot to do with the simplicity of the scene. To compare the effects of raising the voxel resolution, a test is run with the same scene input, but in a $45 \times 28 \times 45$ voxel environment. As figure 8.11 shows the results are different, but it is hard to decide which are the most correct. In a scene this simple the benefit of a more precise motion field is almost lost. Notice however, how the flame becomes very thin in the two examples (most prominent with the higher voxel resolution). As there are no obstacles in the scene, this is actually pretty much what was to be expected, as the setup is comparable to that of a candle burning quietly.

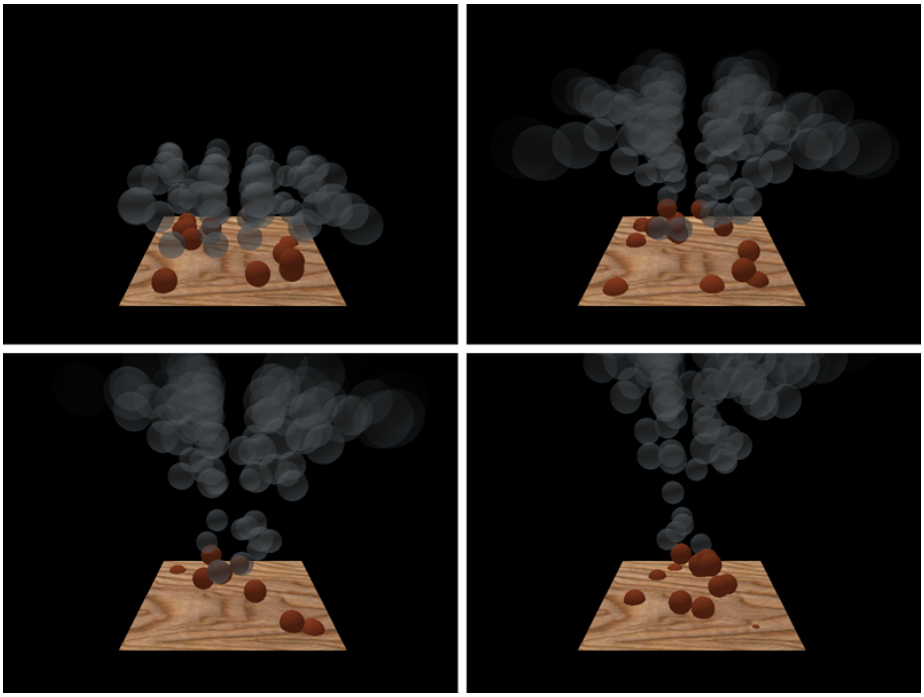


Figure 8.11: *Same setup as in figure 8.10, except that the voxel resolution is now* $45 \times 28 \times 45$. †

Voxel Resolution	200	400	600	800
$15 \times 9 \times 15$	29.8	58.6	88.1	117.2
$45 \times 28 \times 45$	1658.8	3019.8	4375.4	5950.8

Table 8.5: *Rendering times for calculating motion fields in the simple scene from figures 8.10 and 8.11. The motion fields have been iterated 200, 400, 600, and 800 times. The time is measured in seconds.*

In table 8.5 the time used for calculating the motion fields in the above two tests have been compared. The table clearly shows that a lot of time can be saved by selecting a low but still sufficient voxel resolution. At the same time it must be remembered that when creating an animation, the calculation time *per frame* is actually quite low. In the example with the high voxel resolution, the motion field calculation time per iteration is in fact only 7 to 8 seconds, whereas the raytracing time for a picture can easily take several hours when radiosity calculations are included. In section 8.7 we further elaborate on this.

The number of particles used in the scene can be raised to create a more detailed visualisation of the motion field. This has been done in the pictures in figure 8.12, by increasing the *emitter resolution*. As the pictures show, the motion fields seem to be alike in appearance.

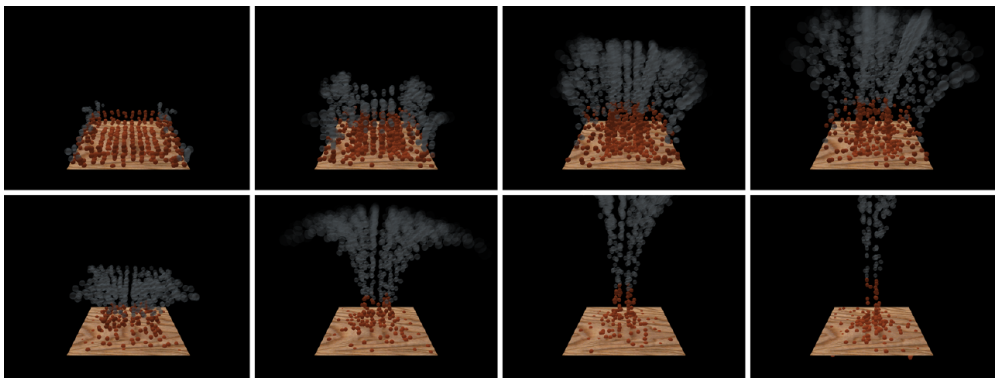


Figure 8.12: *This scene shows the same setup as the ones given in figure 8.10 and 8.11. The only difference is that the amount of particle emitters—and thereby the amount of particles—has been greatly increased. The top row of pictures shows the results obtained by using the small voxel resolution of* $15 \times 9 \times 15$ *voxels. In the bottom row of pictures, the larger voxel resolution has been used. The scene is defined in* `testmotion2.tsr`. †

The number of particle emitters is normally increased to get more detailed results. But the cost might be a big increase in raytracing time. This really becomes a problem when blobs are used instead of spheres to visualise the particles, as blobs tend to slow down the raytracing process greatly. As stated earlier (for instance in section 8.2), backwards warping of blobs can be used to simulate many blobs, using only few blobs. So it should be possible to get very good (and much faster) results by using fewer blobs, but backwarping them more. In figure 8.13 the effect of using blobs instead of spheres is illustrated. The figure illustrates the differences obtained using different amounts of blobs and different amounts of backwarping. The high voxel resolution has been used and the motion fields have been iterated 600 times.

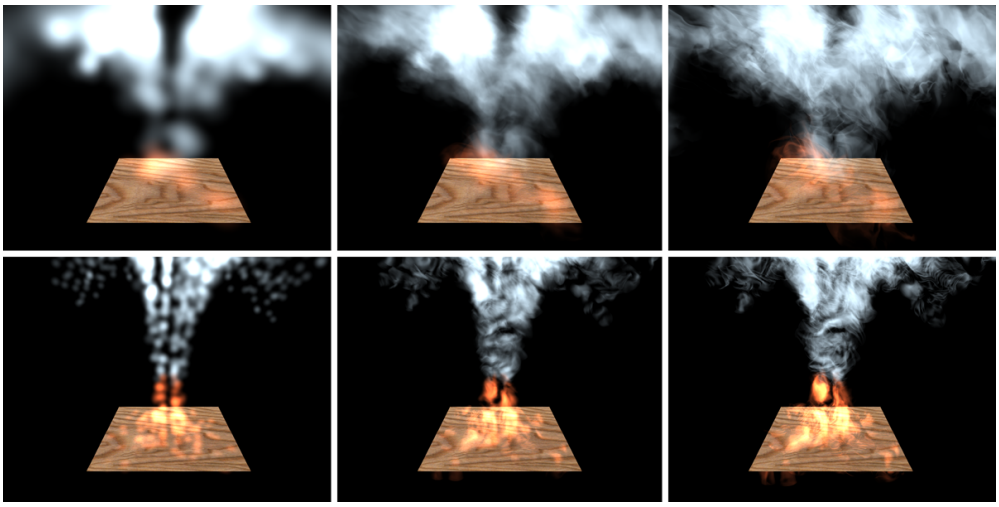


Figure 8.13: *The effect of using blobs to visualise the motion field. The top row of pictures uses 10 fire blobs and 228 smoke blobs whereas the bottom row of pictures uses 151 fire blobs and 1731 smoke blobs. The columns show the results obtained by 0, 5, and 10 backwarp steps.* †

Blobs used	**0 backwarps**	**5 backwarps**	**10 backwarps**
Few	3.2	79.4	268
Many	55.9	79.1	141.8

Table 8.6: *The rendering times of the pictures in figure 8.13 (shown in* minutes*). The rendering time for calculating the motion field has been subtracted, as it is the same for all the setups. Shadowing and radiosity options have been disabled.*

When comparing the pictures of figure 8.13 with the raytracing times of table 8.6 we instantly note the surprising fact that the picture with fewer but bigger blobs actually takes a lot longer to raytrace, when the number of backwarps is large. This is completely unexpected. The only reasonable explanation we can come up with is that as the blobs are bigger they actually take up more pixels in the resulting image, and thus more rays intersect blobs. In the example with many smaller blobs, a lot of overhead time is used to check each blob for intersections with the ray (thereby spending more time for 0 backwarps), but all in all fewer blob intersections exist in the image and thereby the time penalty for raising the number of backwards warps is not as big. This is emphasised by the way the backwarping is handled in the raytracer. When a blob is tested for intersection with a ray, the radius of the blob is first scaled by a factor depending on the number of backwards warp steps. This has to be done, as the warping of a blob might make different volumes of the blob drift apart, thereby increasing the size of the bounding sphere. For the big blobs this increase in size means that each blob will now cover a lot more pixels, and thereby intersect a lot more rays. The smaller blobs will each only cover a few more pixels, when the size is increased. This is illustrated in figure 8.14.

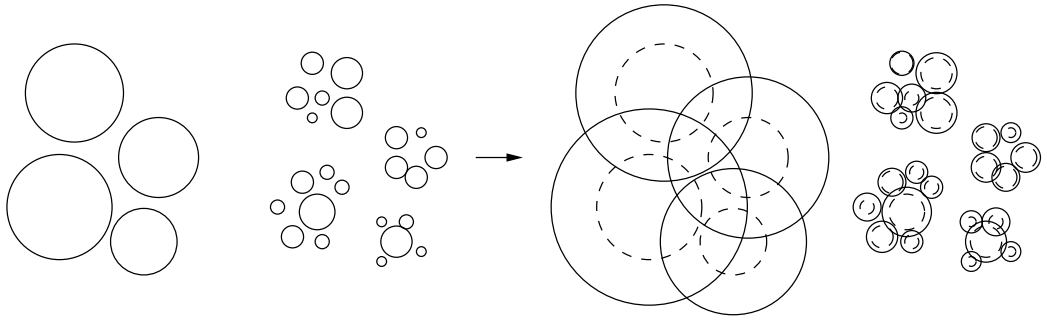


Figure 8.14: *The size increase of backwarped blobs. When backwarping is applied to blobs, the blobs must be intersected using a radius greater than the initial radius of the blobs, to take the diffusion effects into account. This results in more overlapping between blobs and more intersections between rays and blobs.*

8.4.2 A more complex motion field

The automatically calculated motion field incorporates the effects caused by solid objects in the scene. This has not been made clear by the examples shown so far. To present this feature, the scene used in the previous section is modified by adding a floating barrier, hovering above the burning square. The barrier is tilted in a way that should make the fire and smoke drift to the right, as seen from the camera. Furthermore walls are constructed, to make the scene look like a room. This is done to make the picture more interesting. Figure 8.15 shows how the resulting motion field *does* flow to the right of the barrier. Actually the images do not look all that impressive. Especially the picture using spheres instead of blobs looks wrong. The fire and smoke blobs seem to pass through the plate while being only nudged gently to the right. Well, the fire and smoke does pass through the plate at the rightmost edge, but that is not due to any errors. One reason is, that the voxel space representation of the scene is not exactly correct, as the solid voxels are found by shooting rays through the centre of each row of voxels, thus detecting if each voxel contains any solid graphics primitives. This, however, only detects a graphics primitive if it occupies more than half of a voxel, resulting in the above mentioned effect. Also, as the blobs are only moved at the centre, one end of a blob can easily pass through a solid voxel, as long as the centre remains clear.

On the left side of the pictures in figure 8.15 it seems that the smoke just passes through the barrier unhindered. It does not! As an animated sequence of the scene shows (see section 8.6), most of the very big spheres of smoke in the left picture *do* actually pass nicely past the plate. But a few manage to get stuck below the barrier, probably due to the alignment of the solid voxels (as discussed in section 5.5.3, see figure 5.20 on page 73), or numerical errors might have pushed the blobs *inside* the solid voxel, preventing any further movement. A feature implemented in the animation system is that the smoke blobs inflate over time while becoming more and more translucent, thus simulating the slow dispersion of the smoke. So in the pictures the amount of smoke just around the plate is actually just 'old' smoke particles, that have been trapped by the air currents.

In the earlier example with the 'simple' motion field we saw that a safe way to reduce calculation time was to use a smaller voxel resolution. For more complex scenes this should

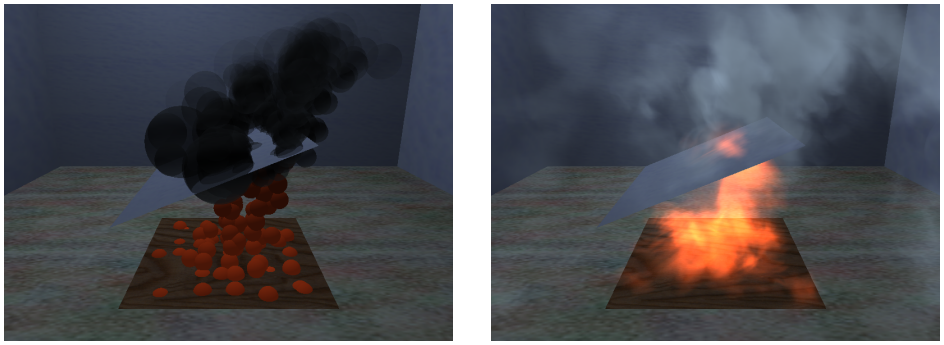


Figure 8.15: *This figure shows how the motion field calculations handle solid objects blocking the fire and smoke. The same picture has been rendered with spheres and blobs used as particles. The voxel resolution is* $30 \times 19 \times 30$. *The scene is created in* `testmotion3.tsr`. †

be done very carefully. As an example look at figure 8.16, where the voxel resolution has been reduced to $10 \times 6 \times 10$. In this example, the tilted floating obstacle lies too close to the burning square (The voxels are so big, that the two squares 'collide' in voxel space). It is clear, that the result is completely wrong, when comparing with the results from figure 8.15.

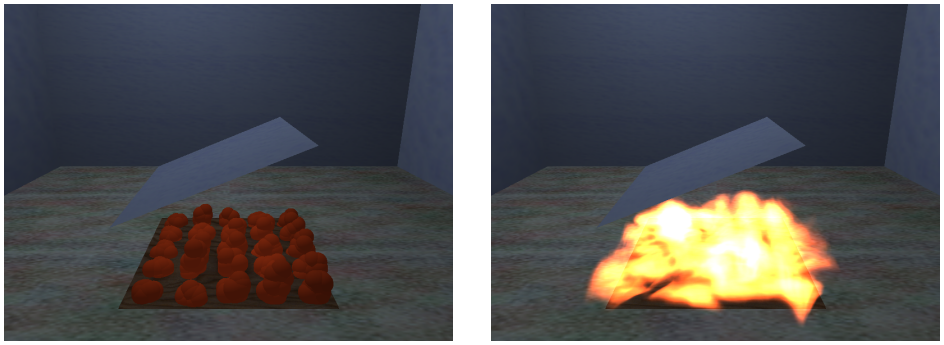


Figure 8.16: *This figure shows the effect of using too small a voxel resolution (*$10 \times 6 \times 10$*). When compared with figure 8.15 it is clear, that the result is wrong.* †

8.5 Second level turbulence

The use of a precalculated second level turbulence (as opposed to the turbulence dynamically calculated in the animated motion field) as a way of adding realistic looking detail to a flame or a smoke cloud, has already been discussed earlier. In this section the *effect* of this turbulence will be shown with a couple of examples. Two parameters are used for controlling the second level turbulence. The first parameter scales the frequency. This affects the rate at which the turbulence changes, but not the amplitude. With this parameter it

is possible to change the turbulence appearance from 'slow' or 'thick' to 'quick' or 'wild'. The other parameter scales the amplitude or 'power'. This parameter decides how big the effect of adding the turbulence will be.

The examples in this section have been created using the same scene as in figure 8.15. The picture to the right in this figure shows the turbulence that is added as a default by the raytracer. We now change the parameters of that turbulence in the following ways:

(a) The second level turbulence is completely disabled.

(b) The turbulence is made 'quicker'.

(c) The turbulence is made more 'powerful'

(d) The turbulence is made both 'quicker' and more 'powerful'

(e) The turbulence is made 'slower' but more 'powerful'

Example *a* is especially interesting, as it shows only the turbulence created by the animated motion field. Any turbulent features of this example arise from calculations based on the motion of hot gas in the scene (as described in section 5.5.1). This is in other words the best we can get without 'cheating' by adding random turbulent noise. Figure 8.17 shows the result.

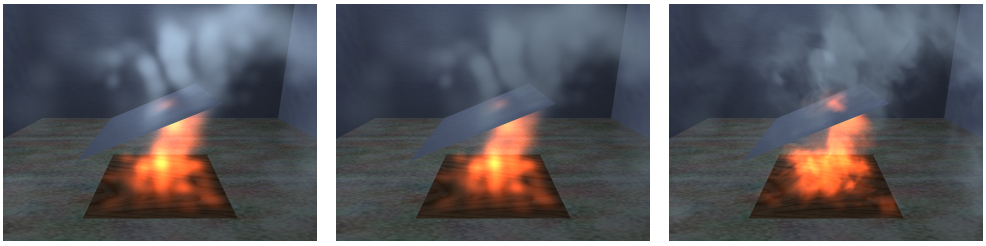
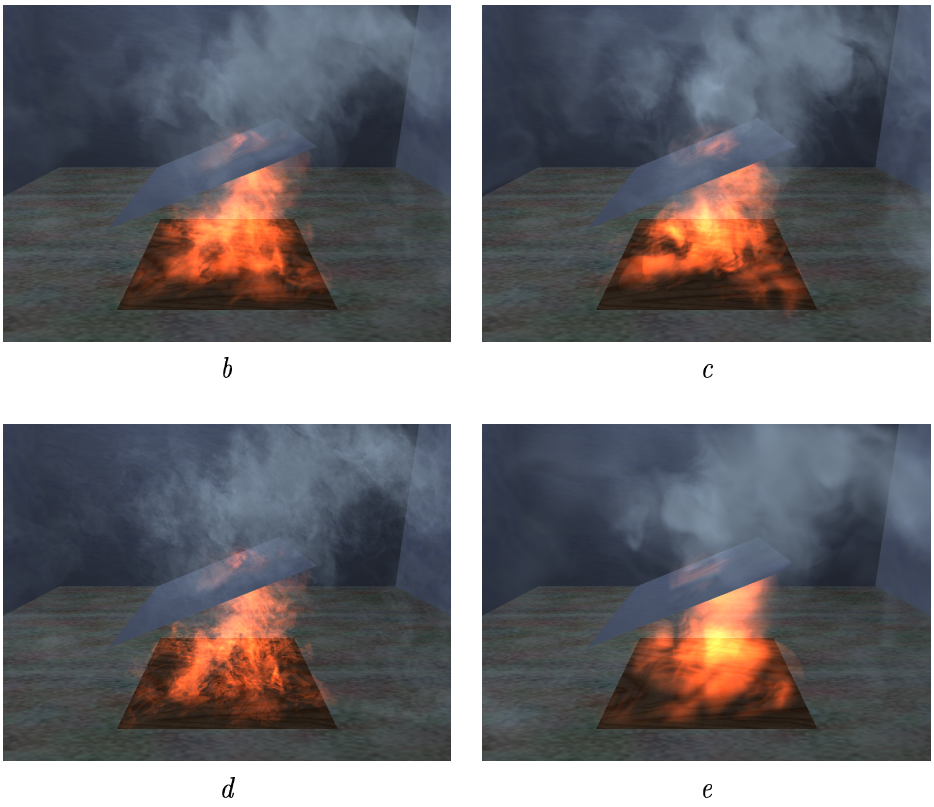


Figure 8.17: *The effect of second level turbulence. The picture on the right is from figure 8.15. To the left is the exact same scene, but rendered without any backward warping of blobs. The picture in the middle is example* a*, where the animated motion field is used for backwarping but the secondary turbulence is disabled.* †

The figure shows—dissapointingly—that in this setup it is very difficult to detect any differences between the picture with no backwarping performed, and the picture where the animated motion field is used for backwards warping. The only noticeable difference is, that the smoke is somehow less translucent when no backwarping is performed, and that is probably caused by the sample algorithm and not by the backwarping itself. The reason for this lack of results must be, that the animated motion field is too simple to add much detail, or that the values in the field might be too small to create any noticeable change in the big blobs used in this scene. A way to improve this might be to scale the animated motion field when used for backwarping, just as the precalculated second level turbulence field can be scaled (as shown shortly). On the other hand, a more complex scene or a more detailed voxel space representation might also do the trick.

The results obtained by running examples *b* - *e* are shown in figure 8.18.



b *c*

d *e*

Figure 8.18: *The effects of changing the parameters for the second level turbulence are shown in these four pictures. They are all modifications of the setup of figure 8.15. The top left picture* b *has quicker turbulence. Picture* c *to the top right has more powerful turbulence. The bottom left picture* d *has both quicker and more powerful turbulence, and the last picture* e *has slower but more powerful turbulence.* †

The pictures *b* and *c* have almost the same appearance even though the first one has 'quicker' turbulence, whereas the second has more 'powerful' turbulence. In both pictures the modifications to the parameters give the visual appearance of 'more' turbulence, but it is difficult to see exactly how this is accomplished. When looking at picture *d*, however, it is very clear that the turbulence is *both* 'quicker' and more 'powerful'. Here the two effects enhance each other, resulting in a very detailed but also very unrealistic result. In picture *e* the turbulence has been made more 'slow'. The flame and smoke appear to be thicker here.

An interesting side effect of raising or lowering the 'quickness' of the turbulence is, that a slow turbulent flame appears looks like a smaller flame situated closer than a very quick turbulent flame. This can be seen when comparing picture *d* and *e*, where the latter (in our opinion) looks very small and calm opposed to the former that look like some very wild fire effect very far away.

8.6 Animations

In this section we discuss a few sequences we have created of the animation of fire. The main factor in an animation, that can not be seen in single still pictures, is the look and feel of the animation of fire (and smoke) particles. The results obtained using an animated motion field for calculating single pictures have been shown in section 8.4. This section shows how the scene changes over time.

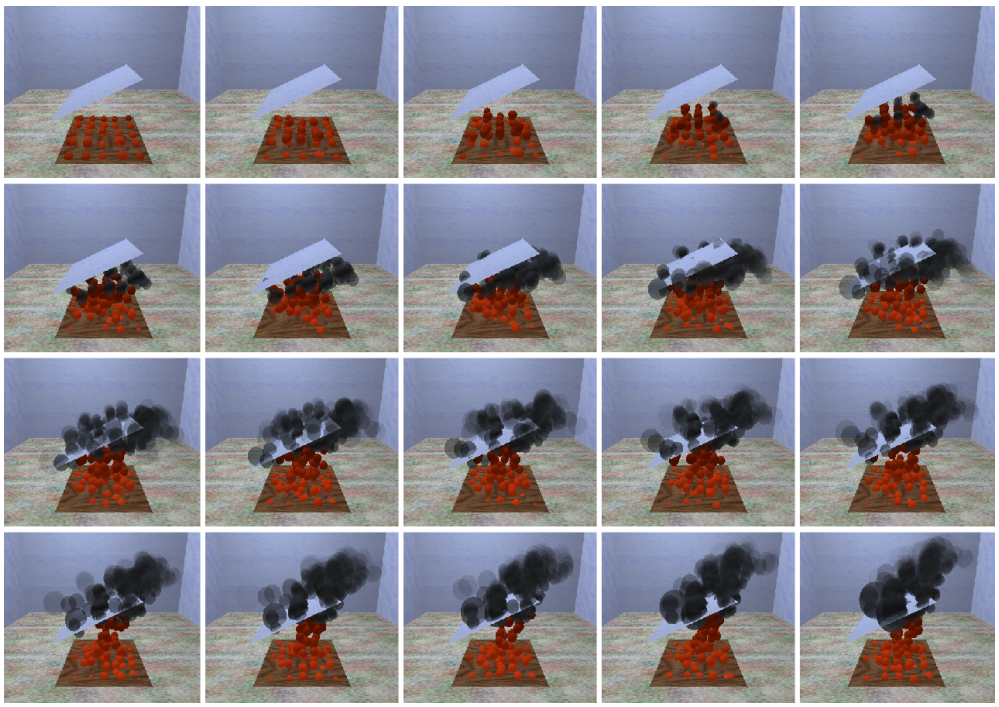


Figure 8.19: *Animation showing how the scene from figure 8.15 evolves from the first frame until frame 1000. Every 50'th frame has been rendered. The animation shows how some smoke particles are caught beneath the barrier.* †

Motion field

In figure 8.15 a scene was shown where spheres were used as a tool for visualising fire and smoke visualising of blobs. It was argued here, that the big smoke spheres around the barrier (in the image on the left of the figure) were caused by smoke particles being caught beneath the barrier. We have created an animated sequence of this setup which shows that this is in fact the case. The sequence can be seen in figure 8.19.

As a more convincing example of how the animated motion field creates a realistic wind field is our 'twisted chimney' scene shown in figure 8.20. The scene shows a small burning plate in a room with a twisted chimney in the ceiling. The front walls of the room and chimney have been made invisible, so that it is possible to see what happens inside the room. But the walls *are* there, and are being used as solid objects in the motion field algorithm. The fire has been rendered using spheres instead of blobs to more easily show how the smoke particles are being animated.

The figure clearly shows how the hot gas rises, enters the chimney, and then follow the shape of the chimney upwards to the top of the image. In the beginning some of the smoke misses the chimney and takes a detour down along the right wall and out of the picture in the bottom of the screen. This happens because the room has no floor.

When a floor is added to the scene the fire and smoke have problems rising, as the motion field algorithm ensures that no volume of gas can suddenly rise without another volume of gas rushing in to take its place. The surprising result is, that the smoke is being pushed towards the floor, where it 'crawls' along until reaching the walls. Then it 'climbs' the walls, 'crawls' along the ceiling, before finally escaping through the chimney. Without a floor, gas can rush in from below the fire, ensuring a more 'correct' behaviour.

It should be possible to create a more correct animated motion field by ensuring that fire is allowed to rise *no matter what*! We will discuss this more deeply on page 137.

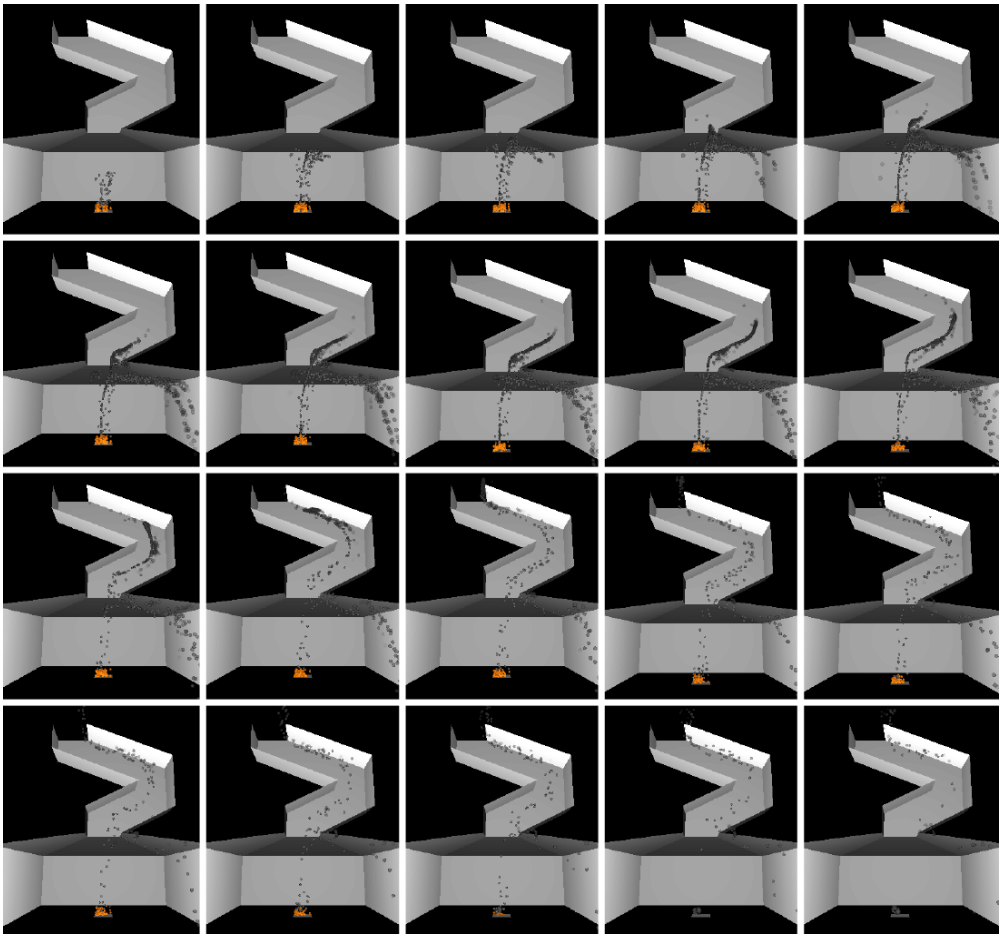


Figure 8.20: *Smoke escaping the scene through a very strange chimney. Another example of the animated motion field in action. The scene is defined in* `chimney.tsr`.

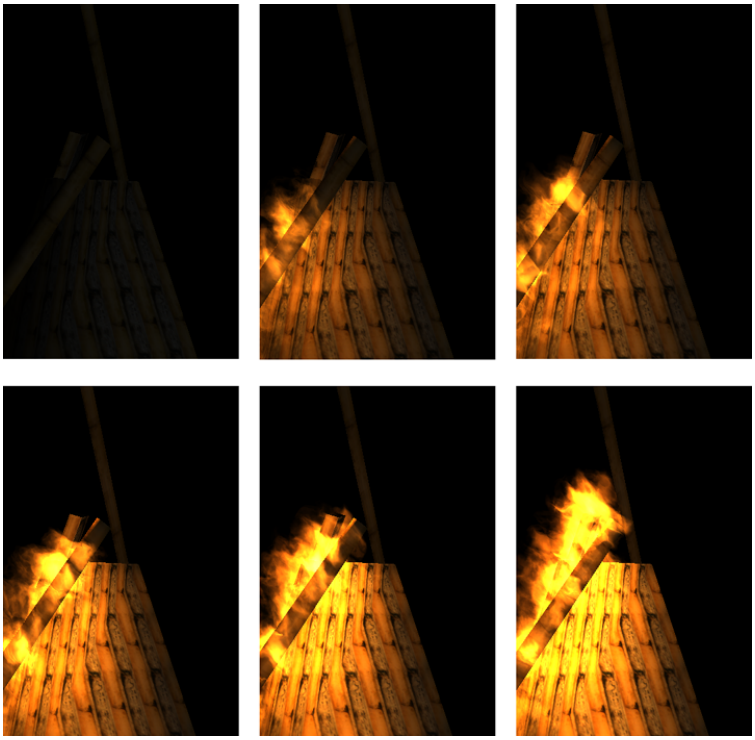


Figure 8.21: *Animation showing the ignition of a bonfire. The animation has been stopped early as the spread of fire quickly resulted in an inferno of light, that was not very pleasing to look at. This shows the difficulty of choosing a good setup. The scene is defined in* `bonfire.tsr`.

Spread of fire

As an object burns it generates heat, that is used for creating the animated motion field. In the motion field algorithm the heat is spread throughout the scene through convection and radiance. A temperature field is maintained so that the temperature in any part of the scene is known at any time. This temperature field can be used to control the spread of fire. As discussed in section 7.3.2 fire particles are created by a grid of particle emitters spread out on each combustible object. Each particle emitter can then use the temperature field to decide whether or not it is warm enough to emit fire particles. An animation showing this is given in figure 8.21.

The figure shows a pile of combustible material (wood) being lighted in the lower left corner of the pile. The sequence of images then illustrate how the fire spreads to engulf the entire wood pile. Later on in the sequence, the fire becomes so wild, that it lights up the whole scene to a degree that makes the result completely unusable. This just goes to show how difficult it can be to select the correct fire colour, rate of birth, emitter resolution, and so forth.

Fuel map

We have earlier discussed how fuel maps could be used to create interesting effects (sections 5.3 and 7.3.2). Our raytracer implementation can use very simple fuel maps to decide which

parts of a surface are combustible. This has been used to create the burning 'THESIS' logo on figure A.3 in appendix A.

We have created an animation using a spiral shaped fuel map. The scene is heated at one end of the spiral allowing fire to start. Heat is then propagated throughout the scene by the animated motion field, resulting in a spread of fire along the spiral. Figure 8.22 shows the result.

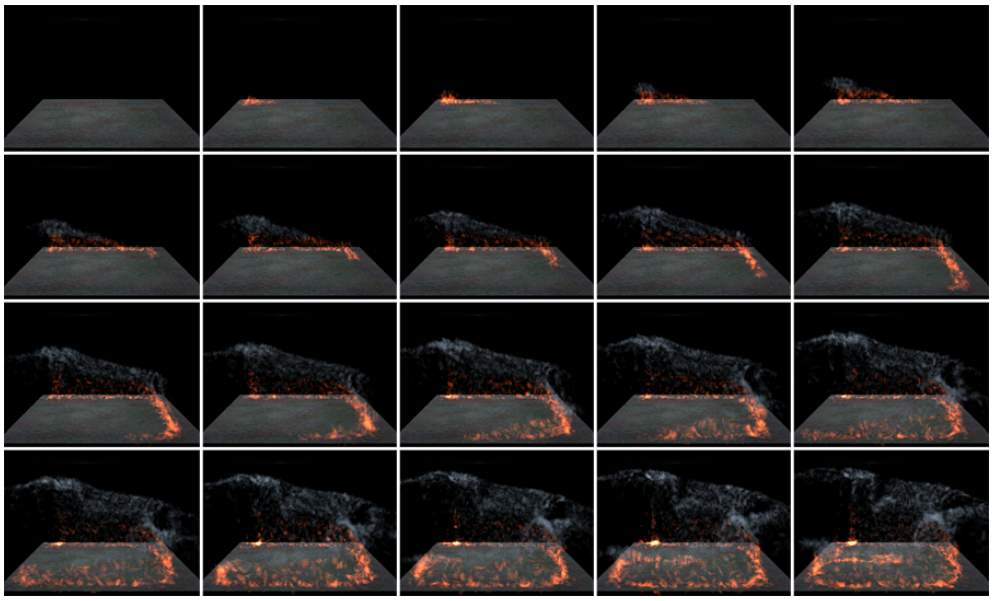


Figure 8.22: *Animation showing the ignition of a spiral shaped fuel map on a surface. The fuel is ignited in the upper left corner and the heat is then propagated by our motion field calculations to heat up the entire spiral. The scene is defined in* `testfuelmap.tsr`. †

8.6.1 General observations

Some of the example animations of this section are samples of small 25 frames per second movies that we have rendered. These movie clips show what is impossible to detect from the still pictures of this section: There is something wrong with the speed of the flames! They seem to move much too slow when compared to the smoke. There are a couple of reasons for this.

First, as we saw in section 8.5, the visual influence of the precalculated second level turbulence is much stronger than the influence of the animated motion field. This means, that many of the recognisable features in the images come from the second level turbulence. These features are made to rise slowly, to give the appearance of rising smoke, but that way this appearance affects the fire as well.

Second, when something burns in our raytracer, the object gets hot and starts emitting fire blobs. Because of the heat, the gas in the area rises, carrying the fire blob along. This is not quite the way most fires behave. A much more accurate model would be one where the burning objects release gas (fuel) with some force along some vector (most often up). The speed of the gas could well exceed the speed caused by thermal buoyancy, and the algorithm for calculating the animated motion field (section 5.5.1) should not be able

to change this speed. This algorithm takes an initial guess at the motion vectors and iterates until a solution is found where gas does not disappear from the scene. As can be seen from figure 5.17 on page 70, a fast jet of gas from one point in the scene will be slowed by the iterative algorithm. In fact, the algorithm needs to be expanded to handle 'unmodifiable gas jets', and a strategy should be found for choosing the velocities on the surfaces of burning objects. These modifications seem to be fairly easy to implement, but care should be taken when handling 'unmodifiable gas jets', as these might lead to instability or convergence problems (just think of the setup where gas jets are blowing into the same voxel from all sides.).

8.7 Image quality vs. rendering times

So far a good deal of energy has been put into describing how time consuming the specific features of the raytracer are. In this section we will try to show how the resulting pictures improve when adding more and more detail, and how rendering time increases at the same time. We choose the scene from section 8.5 as the basis for our refinements, using 1000 iterations of the motion field. To illustrate the connection between the number of blobs and the rendering time, a version of the scene is added, where the number of particle emitters is increased, which, of course, increases the number of blobs.

The two scenes are first visualised using very few of the raytracing features. Both radiosity and shadows are disabled, and the voxel resolution is kept small. The results are shown in figure 8.23.

As a next step, the voxel resolution is increased, to create a more convincing motion of flame and smoke resulting in figure 8.24. This will cause an increase in time usage for calculating the motion field, but the time used for the actual raytracing will remain the same. Table 8.7 shows the time used for calculating the motion fields. The rest of the test examples in this section uses the high voxel resolution (which is $30 \times 19 \times 30$), and all the time measurements have been modified to exclude the time used for calculating the motion field in each test. These results can be found in table 8.8 on page 142.

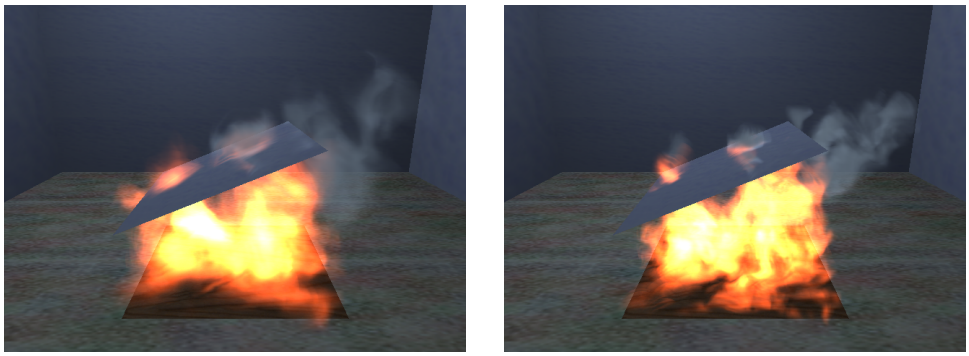


Figure 8.23: *A simple raytracing of a scene. In the scene to the right more blobs are used. The pictures are raytraced using no shadows, no radiosity and a voxel resolution of* $15 \times 9 \times 15$. *The scenes are described by the two files:* `testmotion3.tsr` *and* `testmotion4.tsr`. †

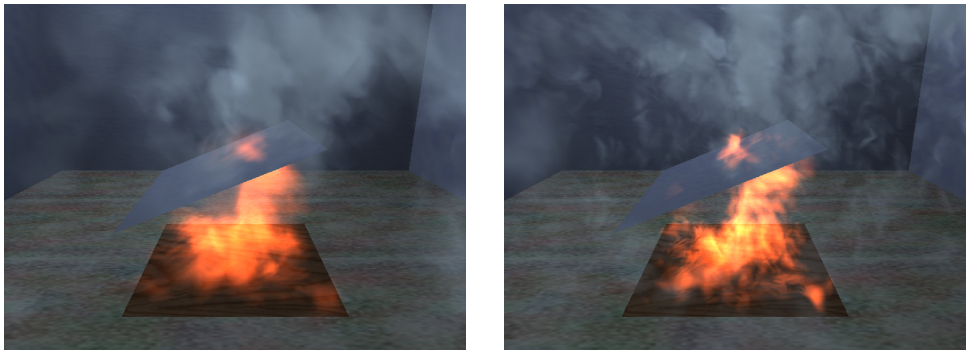


Figure 8.24: *In this figure the voxel resolution is changed to* $30 \times 19 \times 30$. *There is still no shadowing or radiosity.* †

Voxel resolution	**Calculation time**
$15 \times 9 \times 15$	2.6
$30 \times 19 \times 30$	31.2

Table 8.7: *This table shows the time usage for calculating some of the motion fields used in this section. More precisely the motion field used for the scene with 113 fire blobs and 499 smoke blobs. 1000 steps of iteration have been applied. The time is measured in minutes.*

We will now start adding rendering details to the images. First the radiosity calculations are added as shown in figure 8.25. Now the different levels of shadowing calculations are enabled. First the 'simple shadows' option is tried (figure 8.26). When using 'simple shadows' blobs do *not* cast any shadows, but all other objects do. Next is the 'simple blob shadows' option (Figure 8.27), where the shadowing algorithm is greatly simplified for blobs. Finally, the scenes are raytraced with all shadowing options enabled (Figure 8.28).

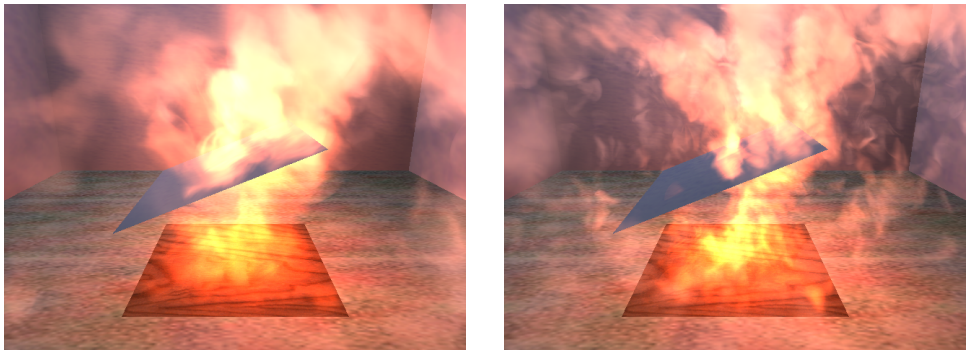


Figure 8.25: *The radiosity calculations have been enabled, but there is still no shadowing.* †

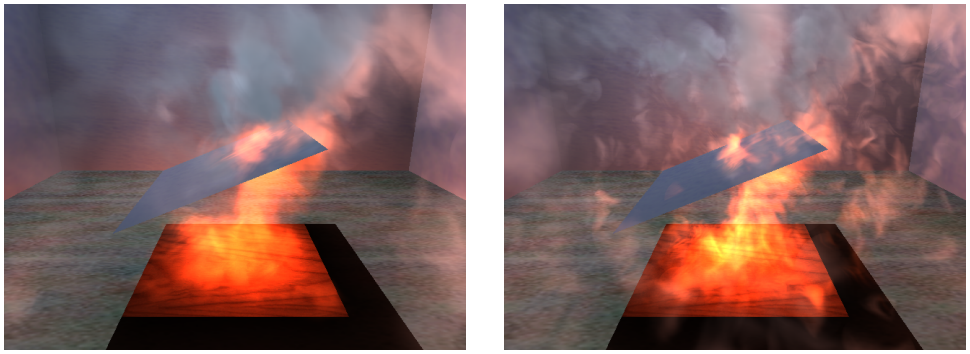


Figure 8.26: *The radiosity calculations are enabled as well as the 'simple shadows' option.* †

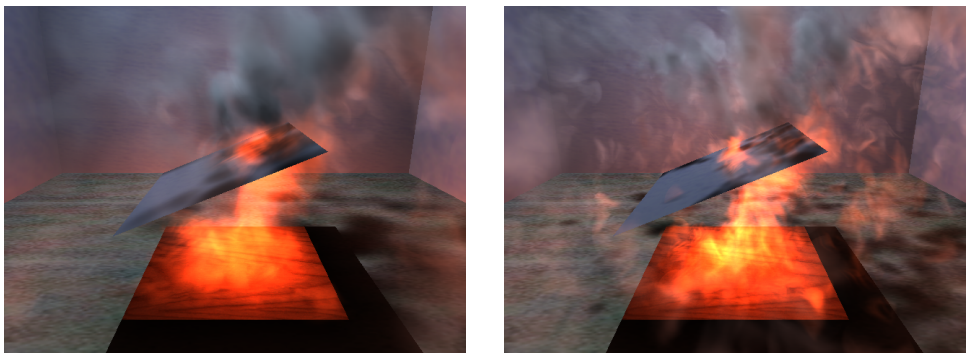


Figure 8.27: *The radiosity calculations are enabled as well as the 'simple blob shadows' option.* †

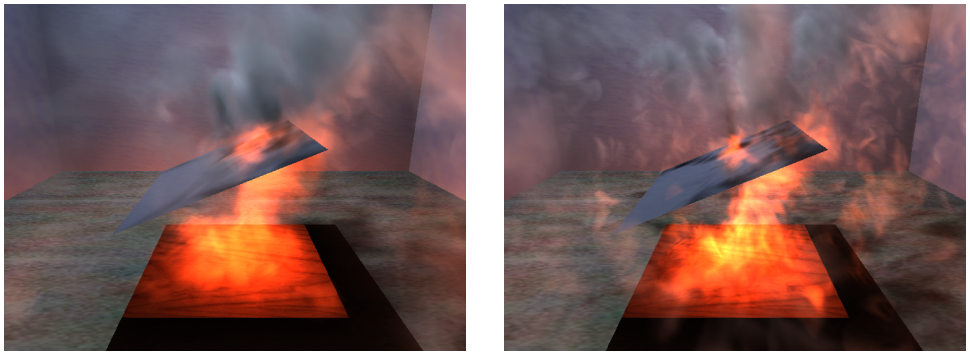


Figure 8.28: *Both the radiosity and the shadowing calculations have been enabled.* †

Notice the somewhat subtle changes in the pictures, as more rendering detail is added. In 8.25, where no shadowing options are enabled, the smoke blobs receive huge amounts of light from the fire blobs, making them all very bright and almost pink. With the simple shadows (fig. 8.26 — see a colour version in Appendix B) enabled it is only those blobs that are not covered by the plate, that become very bright. The rest are lighted in a monotone

grey colour, from a point light source at the upper left part of the room. When simple blob shadows are used (fig. 8.27) the smoke instantly becomes much more interesting. It now casts shadows on the floor, and it is even self shadowing. This way the smoke appears much more 3-dimensional and interesting to look at. Finally, when all rendering details are added (fig. 8.28) the lighting effects become marginally better. This is seen best on the smoke shadows on the floor.

Looking at the time usages shown in table 8.8 it is clear that with both radiosity and full shadowing enabled the raytracer is very slow. Especially the number of blobs have a big effect on rendering time. For that reason the raytracer can be configured to skip radiosity calculations for a lot of blobs. In the examples in figure 8.29, the raytracer only calculates radiosity for one in every 25 blobs. Actually all blobs must still 'receive' light from the radiosity calculations, and all blobs must still be rendered, so the gain should not be enormous. As table 8.8 shows, the gain is something like a factor 2 to 3. The image quality is obviously also reduced, but the result must be said to be quite good. The biggest difference is the amount of shadows on the floor. (As many smoke blobs do not contribute to the radiosity calculations, the floor is lit more strongly by the light source in the upper left of the image.) It is interesting to note that the gain is much bigger when using fewer but larger blobs. In fact the table shows, that doing the full radiosity and shadow calculation is most expensive in the scene where few but big blobs are used. The reason for this is discussed on page 129 (see figure 8.14). When skipping the radiosity calculations for a major part of the blobs this is changed, and the scenes with big overlapping blobs benefit most.

Finally, we will add as much detail as possible in an attempt to make the quality of the resulting picture as good as it can get. We enable both shadowing and radiosity, we allow the radiosity algorithm to complete 5 iterations (instead of the default 2), we backwarp the blobs 10 steps (default is 5) and we sample each blob 9 times (default is 5). The resulting picture is shown in figure 8.30. We have tried to add light reflecting properties to the smoke, (so far the smoke has not reflected light) but this resulted in such a huge increase in radiosity calculation time that it was hopeless to think that the computer would refrain from crashing in the period (We tried! The computer went down after a month—the picture was not done then).

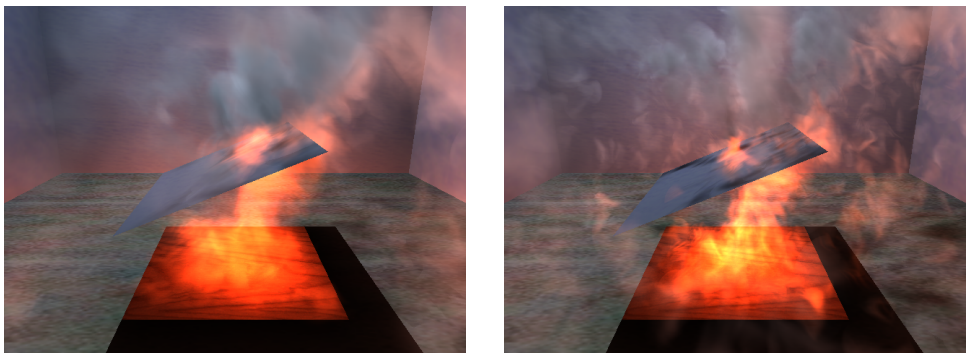


Figure 8.29: *Both radiosity and the shadowing calculations have been enabled. In these pictures the radiosity algorithm only permits every 25th blob to radiate.* †

Figure	612 blobs	2248 blobs
8.24	93.0	90.8
8.25	93.3	91.0
8.26	120.1	201.5
8.27	193.5	501.8
8.28	3071.7	2981.7
8.29	1236.2	1862.1
8.30	—	25778.7

Table 8.8: *The rendering times for the pictures shown in this section. All the pictures were raytraced with a* 640×480 *pixel resolution. In the pictures with fewest blobs there were 113 fire blobs and 499 smoke blobs were created by the fire emitters in the scene. In the other pictures 346 fire blobs and 1902 smoke blobs were created. Note that in this table, the rendering times are measured in* minutes*!*

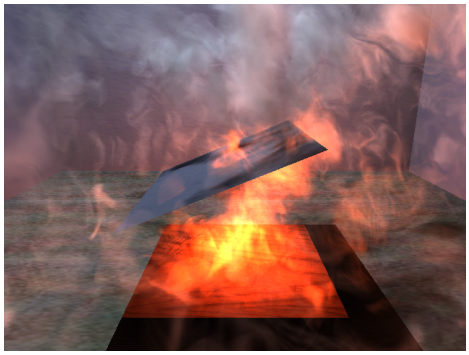


Figure 8.30: *This picture should be of absolute top quality. All features (except smoke reflectivity) have been enabled to add as much detail as possible to the scene.* †

As the tests show, it is advisable to take extra care when selecting the parameters for a given scene. As table 8.8 shows, adding too much detail to the setup can easily make the rendering prohibitively expensive. The trick is to choose the amount of detail that match your needs exactly. In the examples of this section, the use of a full shadow calculation seem to be a waste of time. As an example we have tried to make one more image, where only the simple blob shadowing algorithm is used, radiosity is only calculated for every 50'th blob, the blobs are only sampled using 3 sample steps and are only backwarped in 4 steps. The resulting image is shown in figure 8.31 and was rendered in $80\frac{1}{2}$ minutes (excluding the time used for calculating the motion field). Note, that this is even faster than the version rendered with *no* shadowing and *no* radiosity (figure 8.24). Note also, that this image shows a minor error with the shadows of the blobs. On the floor to the right of the burning plate, the shadows of the smoke blobs create some very clear edges. These are obviously too regular to be anything but errors. We have no idea what causes this, and we have not been able to recreate this error in other pictures. Our best guess is that it has to do with many overlapping blobs being sampled too sparsely.

Our implementation has been kept very simple and unoptimised, but even though a number of optimisations to the algorithms were proposed in section 7.4, it is improbable

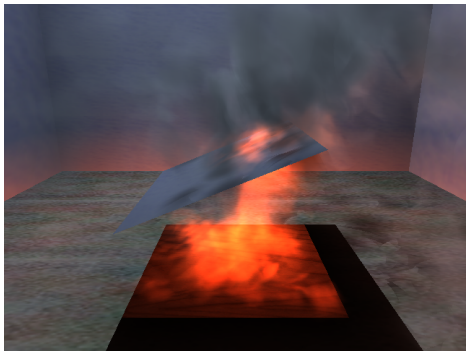


Figure 8.31: *This picture shows our tradeoff between getting as good image quality as possible while still keeping the rendering time moderately low. The picture was rendered in* $80\frac{1}{2}$ *minutes (excluding the time used for calculating the motion field).* †

that they will improve the overall performance by more than a factor 5 or perhaps 10. As a result it will still be possible to create scenes of such complexity, that they can not be rendered in any usable amount of time. This is clearly demonstrated with the last example of table 8.8 which took almost 18 days to render, and that was without adding light reflecting properties to the smoke blobs.

8.8 Simulating Scattering in clouds

Creating clouds is admittedly a bit outside the scope of this thesis, but somehow it is quite exciting to realise, that the models and visualisation techniques described here can be used with small effort to create convincing renderings of effects they were never meant to handle.

In section 3.3.1 we have discussed light reaching a participating media (such as smoke) and scattering inside the media. To capture this it is necessary to solve or approximate the scattering equations—a set of differential equations—which is a computationally expensive task.

[Rushmeier and Torrance, 1987] present a solution called 'the zonal method', where a radiosity algorithm is generalised to deal with emission, scattering and absorption by the participating media. In their solution the media is approximated by a set of rectangular boxes, and all rays (from raytracing or the radiosity algorithm) are sampled through the boxes using a user defined number of sample steps, thus approximating the scattering equations by a discrete summation.

This sounds very much like the way blobs are handled in our raytracing algorithm. The most prominent differences are that our blobs are not in any way box-shaped, and that we have not incorporated the scattering equation. We think that the self shadowing ability of our blobs coupled with the reflectivity of the radiosity model *must* be a very good approximation to the effects described in the scattering equation. If the general radiosity algorithm can be used to provide convincing results these are obtained 'for free' as an unexpected bonus. Furthermore, the participating media will be much more interesting in our setup, as the backwarping of the blobs contribute directly to the calculations.

8.8.1 Clouds

When visualising the effects of scattering and absorption, clouds have often been used as visualisation scenes, so we have created a cloud scene containing 60 blobs. Figure 8.32 shows the resulting cloud, as it is traditionally visualised using no self shadowing and no scattering.

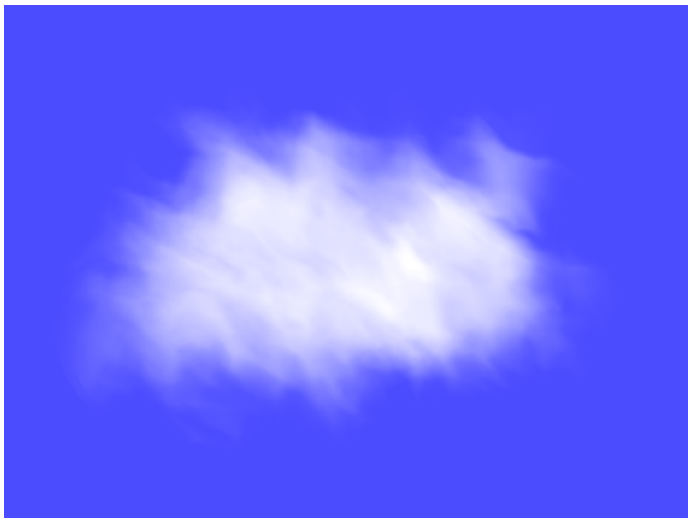


Figure 8.32: *A cloud in the sky. This picture has been rendered using no self shadowing and no radiosity.* †

To simulate the absorption occurring inside a cloud, the self shadowing ability of the blobs is enabled. Normally when calculating absorption, an *absorption coefficient* is set by the user, defining the fraction of light that is absorbed inside the cloud. This coefficient is not available in this approximation, but is simulated by the translucency of the blobs. The scattering is simulated by letting the blobs reflect light using the radiosity algorithm. Again, this has nothing to do with the physics of the real life phenomena, but it gives almost the same visual result. The results can be seen in figure 8.33, where the scene has been visualised both with a light source above and with a light source to the left of the cloud.

Both the shadowing and the reflectivity calculation produce more convincing results the more blobs are used. [Rushmeier and Torrance, 1987] use very few volumes (preferably only one surrounding the whole scene). This is why they *have* to include the scattering equation in the radiosity calculations. As our cloud consist of many blobs, the inter-reflectivity automatically calculated by the radiosity algorithm creates very convincing results. And it is fairly fast too. Only a few seconds are used calculating radiosity in the images of figure 8.33.

Unfortunately, we have not had the time to incorporate the scattering equation either as a stand alone shader or as a part of the radiosity algorithm, so we have not been able to compare rendering times and visual quality of the two different methods. This is another obvious area of further research.

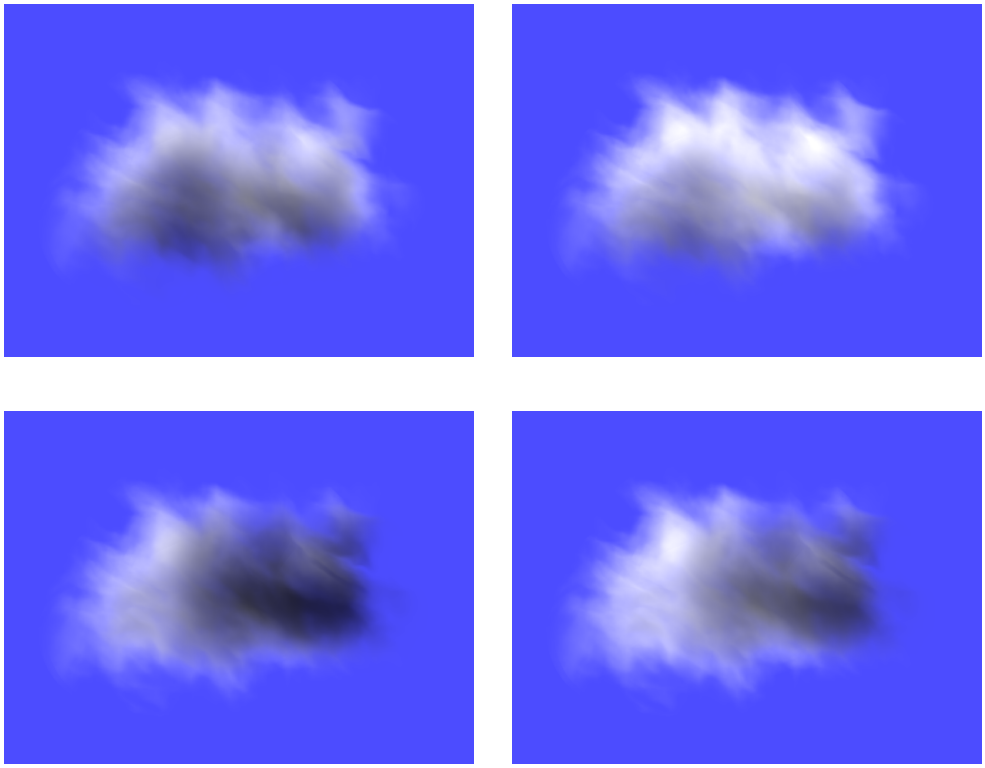


Figure 8.33: *The cloud scene from figure 8.32 has now been visualised using self shadowing and scattering. The images on the left show the effect of adding self shadowing capabilities to the blobs creating the cloud. Notice how dark the cloud gets in the area, that light cannot reach. In the images on the right scattering has been simulated by 10 iterations of the radiosity algorithm. Here the cloud looks much more real. In the top row, the light source is situated above the cloud, and in the bottom row it is situated to the left.* †

Chapter 9

Summary & Conclusion

"...and at the same time keep[ing] scene modelling simple and image generation economical, we have adopted the strategy of the impressionist painters: to represent the essence of a scene as efficiently as possible"

Geoffrey Y. Gardner

In this thesis we have attempted to give an overview of computer generated fire and smoke; how the techniques work, what the historical background is, what present-day applications and implementations offer, what potentials exist in the field and finally we have tested several methods of replicating fire digitally, both in two and three dimensions. We have made several 2D and $2\frac{1}{2}$D implementations and our own raytracer featuring methods that we think are the most promising.

9.1 Completion of our goals

Right from the beginning we had high hopes. Soon, almost as a surprise, it became clear that this was not at all an easy task as fire can be modelled and rendered in so many ways.

Rendering of fire is an *extremely* difficult task, and to obtain the perfect set of pictures, today, an enormous task lies ahead just fiddling about with the vast amounts of parameters.

9.1.1 Availability of digital fire

In our search for information on digital fire we have not come upon a source like this thesis, nor have we found anything that comes close. All the literature that we have come across is either inadequate or extremely irrelevant for our purposes, which leaves the reader unsatisfied in both cases. We believe that the availability of information on digital fire certainly has been increased by this thesis. We hope that it will provide people interested in digital fire with a good starting point for making digital fire themselves, and a good starting point for finding related literature as well.

9.1.2 Our raytracer

We have made quite an effort to make an application which is a good modelling, animation and visualisation tool. The program has not been optimised for speed.

We have made some very nice animations of fire—in our own humble opinion—when compared to the top-of-the-line commercially available products today. We have made interaction with solid objects possible within our data model. To our knowledge this feature is not available in any commercial rendering application. Improvements in this field could be a hybrid between the upper level turbulence modelling made by [Stam, 1995] and the improved, more stable version of [Foster and Metaxas, 1997b].

9.1.3 Complexity analysis

We have given some comparisons of the possible complexity variations offered by our raytracer. These have generally shown what we expected. The only occurrence of an

unexpected result is the comparison of rendering time using many small blobs versus a few large blobs. The explanation for this is quite simple.

We have had a general problem doing an extensive comparative complexity analysis since there only exists very few algorithms related to the subject, hence nothing to compare with in lack of similar algorithms. We would have liked to present a lot of comparable methods and algorithms, but we cannot do so.

9.1.4 Limited amount of parameters

We have constructed a general model of fire and smoke (which takes the constraints imposed by the local environment into account) that is quite easy to animate without much knowledge of how fire and smoke work or are modelled, animated and visualised. We have furthermore created a solution that eases the control of the spread of fire. This is done without spending enormous amounts of time fiddling with the parameters.

On the other hand the amount of parameters could be increased considerably. We have ignored features like: Effects caused by available oxygen, variations of flammable compounds, and moving solid objects causing disturbance in the Navier-Stokes controlled voxel environment. These features have been replaced with invariants, all of which would add more parameters for the user to control.

The RGB model could be replaced by a more realistic model to remove the inconvenience of the RGB colour model. This would add a much more complex definition of materials for the scene objects, lighting, radiosity, compound emanation and so forth, but would not reduce the number of available parameters.

Real flammable materials could be simulated (or emulated) to remake the incomplete combustion caused by real compounds. This would facilitate a much more general model of black bodies, so that flames could be of any colour. This should of course be used in correspondence with another colour model than RGB. Alas, this would *not* bring down the number of adjustable parameters.

The general idea is of course to have all these features, not to exclude them because they are hard to understand and use, but represented by a simple general model, so that everybody can make digital fire.

9.2 Issues to resolve

Not everything goes as planned. It is difficult to meet the expectations of realism. We had high expectations from the beginning of this project, but we must acknowledge the fact that our fire and smoke does not behave as in real life. In many ways the phenomenon behaviour and milieu constraints that we have obtained are the best of the methods available today. The lack of realism lies in visual artifacts and motion:

Lack of visual realism. We have not achieved a rendering of fire which is perfect. Unfortunately our fire and smoke blobs can penetrate planes. This lies in the nature of the extension of the blob. Nevertheless it is quite unsatisfactory. This unwanted feature could be removed by a bit more costly raytracing algorithm that only considers each sampling point in a blob if it lies on the same side of an intersecting plane as the blob center.

Motion. The fire moves too slow (section 8.6) or the smoke too fast. In order to produce our animations we have been forced to use every nth frame instead of all frames (n lies somewhere between 2 and 10). Generally this is only a timing problem, which is not that hard to find a solution to. What is needed is some information on how fast the flame tongues move vertically and horizontally in every type of fire (as categorised in section 3.1). With this knowledge it will be much easier to time the flames—also in relation to the turbulence field and advection. The fire is considered to behave like a hot turbulent gas, and thus moves with approximately the same speed as smoke. But in an animation this looks wrong. The obvious solution would be to boost the speed of fire more than of smoke.

Backwarping the animated motion field. We had very little success achieving a turbulent visualisation using only backwarping of the animated motion field from section 5.5.1. The animated motion field simply did not contribute any noticeable detail. It should be possible to change this by scaling the motion vectors of the field during the backwarping.

Capturing the qualia of fire. *Qualia* is the property that makes a thing or term different from all others, while describing it perfectly. The qualia of 'red' is 'redness', so the qualia of fire must be "fireness". In order to capture the qualia of fire we have made a big effort to describe fire in detail—both the physical and visual characteristics. Despite this, we find that we have had too little success in really defining this "fireness".

9.3 Future research

In the previous chapters we have mentioned related areas of interest that we have not thoroughly dealt with in this thesis. In this section we summarise possible avenues for future research or enhancement.

Motion blur How it this done with fire? Is there any way that this could be made simple with this kind of object or phenomenon?

Depth of field Is there a way of reproducing this effect easily with artificial fire? As with *motion blur*, it should be researched if there exists simple ways of reproducing this effect.

Physics based colours of fire Examine fire colour properties of existing materials to create more 'correct' flame colours depending on the fuel and the burning substance. This will demand a model of the entire colour spectrum, instead of only the RGB components.

The matter of oxygen It would be interesting to examine oxygen and its implications on blowing fire out or making it burn more intensively because it changes the vapour density and vapour pressure (see figure 3.1 on page 23).

Incorporating correct scattering Incorporate the scattering equations into our radiosity calculations for blobs. It would good to consult [Rushmeier and Torrance, 1987], who do this for rectangular box-shaped volumes.

Fractal fire Fractal fire is still uncharted territory. This should be looked into since many of the results presented on the internet do look promising. Fractals have been used for clouds and smoke, but should be adaptable to fire.

Gardner's Texture function [Gardner, 1985] has made clouds with his texture function. Could it be changed to render fire?

Advanced fuel maps An improvement of fuel maps concerning how fuel is consumed or emitted from the fuel map is needed to gain more realistic animations.

The FFT turbulence The problem with the inverse Fourier transformed turbulence is that it is difficult to animate, i.e. it is hard to parameterise the interval from 'light wind' to 'hurricane' turbulence. Is scaling the simple solution to this or should more complex methods be adapted?

Movable solid objects It would be interesting to examine the handling of movable solid objects more deeply than in section 5.5.3.

More correct motion fields A more detailed model of the motion of fire plumes is described in [Baum et al., 1994] (only in 2D), [Baum et al., 1996], [Baum et al., 1997], and [Mell et al., 1996]. These also include a model of the chemical processes occurring in fire.

Realistic objects In real life objects are coloured by soot, they tend to glow as they burn, and they are reduced to ashes by the burning process. It would be very interesting to incorporate these (and other) features in our models.

Pressure blows from explosions Pressure blows and shock wave are not handled in this thesis [Landau and Lifshitz, 1995] have a very theoretical but extremely thorough chapter on this.

Procedural models Parametric patch modelled or procedurally defined fire could be implemented as done in [Kajiya, 1982] and [Kajiya, 1983].

Rendering methods Scanline rendering of fire is yet to be researched. A lot of research has been done on volume rendering: [Perlin and Hoffert, 1989] has shown that hypertexture objects can be used to model fire balls, and in [Perlin, 1985] a two dimensional solar corona was modelled the same way. [Ebert and Parent, 1990] have rendered gaseous phenomena using the A-buffer that [Carpenter, 1984] proposed.

Shimmering effect How can blobs be expanded to handle shimmering effects caused by heat? We wonder if this could be combined with *photon maps* as discussed in [Jensen and Christensen, 1998]?

Implicit surfaces Implicitly sampled surfaces and polygon fire could be interesting to look into. Most possibly this is one of the areas where there is room for enhancement.

Heat reflectivity Incorporation of the inter reflection of heat between blobs and surfaces in the radiosity model.

Partial evaluation It would be interesting to do research on partial evaluation of the rendering of fire. [Mogensen, 1986] has obtained an eight fold speedup on a simple raytracer.

Fire fighting The research of fire at very high level is carried out extensively by fire fighting companies. The need for visualisation in simulation tools is present and a combination of their knowledge and the methods discussed in this thesis, we think could prove fruitful for all parts. [Fire-fighting, 1999] is a large web-site knowledge database on fire. A lot of related material can be found there, ranging from manageable papers to papers on very accurate chemical and physical models. The research of fire retardants is perhaps not one of the most interesting areas compared to the scope of this thesis, but other areas—such as spread of fire—are researched in depth presently.

Unread literature [Galea, 1997] is a collection of lecture notes that we have tried to obtain but in vain as the author at University of Greenwich nor their publication department have sent us any reply on our requests. From the abstract we think it is highly relevant which makes it even more irritating that it is "unobtainable".

9.3.1 Related work

We have read quite a lot of literature to reach an understanding of (rendering of) fire and smoke. Some of these articles are not discussed in this thesis because they are only slightly relevant or somehow do not fit in. To do them justice, they are listed here:

- [Kajiya and von Herzen, 1984] deals with raytracing of volume densities that covers self shading, light scattering, albedo, and particle systems.
- [Sims, 1990] has developed a language for animation and rendering of particle systems on a parallel computing system.
- [Baum et al., 1994], [Baum et al., 1996], [Baum et al., 1997], and [Mell et al., 1996] use computational fluid dynamics to calculate realistic fire plumes. These can be seen as enhancements of the motion field calculations presented in section 5.5.1, including a much more detailed incorporation of the chemical processes involved.
- [Bukowski, 1996] and [Jones, 1993] describe a modelling tool used by engineers to test the fire safety of buildings. This tool focuses on the spread of fire and contains no fancy visualisations, but indicates a need for three dimansional applications.
- [Ahmed et al., 1994] give a very detailed description of the spread of fire. This description is only valid for vertical and horizontal surfaces.

9.4 Contributions

We have tried a lot of theories and we have discussed a lot of literature concerning fire and participating media. We have taken some of the most promising methods available in the field and combined them into a one-solution-package.

We have developed a new approximation for albedo and scattering of light in clouds and smoke. The force of this method is that it uses a standard implementation of radiosity. Usually the calculation of the albedo and scattering demands an extensive solution of a set of differential equations. Our solution is an approximation, but we think the results look as good as those presented in [Kajiya and von Herzen, 1984], [Rushmeier and Torrance, 1987], [Stam, 1991], [Stam, 1995], and [Jensen and Christensen, 1998].

Furthermore we have combined two of the most recent and most promising methods for modelling, animation, and visualisation of fire and smoke, namely the models presented by [Stam, 1991] and [Foster and Metaxas, 1997b]. While doing this we have discovered a problem with the convergence of the iterative calculation of pressure fields proposed by Foster and Metaxas. They have acknowledged the problem. In our combination of the methods we have used blob particle systems to represent the macroscopic level and backwards warping to model the microscopical level of the phenomena. The animated motion field has been used to model the large scale advection of the blobs and thereby the large features of the phenomenon.

We have succeeding in limiting the amount of parameters needed to model a fire. In fact, by the use of default values, it is possible to define a burning surface in our raytracer simply by stating that the surface is combustible and that it is hot enough to burn. The other parameters used to control the fire all have default values that work well in most cases. These parameters include: Temperature, fire point, colour, fuel maps, second level turbulence scaling, and the rate of birth and size of fire particles.

Lastly we find that we have succeeded in increasing the availability of methods and considerations concerning digital fire, thus giving people interested in this field a good starting point for further research.

Bibliography

[Ahmed et al., 1994] Gamal N. Ahmed, Mark A. Dietenberger, and Walter W. Jones. Calculating flame spread on horizontal and vertical surfaces. *Building and Fire Research Laboratory, NIST, Gaithersburg, Maryland*, April 1994.

[Barnsley, 1993] Michael F. Barnsley. *Fractals Everywhere.* Academic Press Professional, 2 edition, 1988, 1993. Fire Fractal co-author: Laurie Reuter.

[Baum et al., 1989] D. R. Baum, Holly E. Rushmeier, and J. M. Winget. Improving radiosity through the use of analytically determined form factors. *Computer Graphics*, 23(3):325–34, 1989. Proceedings SIGGRAPH 1989.

[Baum et al., 1994] Howard R. Baum, Kevin B. McGrattan, and Ronald G. Rehm. Mathematical modeling and computer simulation of fire phenomena. *Fire Safety Science - 4th International Symposium*, Proceedings:185–193, June 1994.

[Baum et al., 1996] Howard R. Baum, Kevin B. McGrattan, and Ronald G. Rehm. Large eddy simulations of smoke movement in three dimensions. *Proceeding to Interflam '96*, pages 189–198, March 1996.

[Baum et al., 1997] Howard R. Baum, Kevin B. McGrattan, and Ronald G. Rehm. Three dimensional simulations of fire plume dynamics. *Fire Safety Science*, Proceedings of the Fifth International Symposium:511–522, 1997.

[Blinn, 1982a] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics (SIGGRAPH 1982)*, 16(3):21–29, 1982.

[Blinn, 1982b] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Computer Graphics*, 1982.

[Brigham, 1988] E. Oran Brigham. *The Fast Fourier Transform and its Applications.* Prentice Hall, 1988.

[Bukowski, 1996] Richard W. Bukowski. Applications for fastlite. *Computer Applications in Fire Protection Engineering Technical Symposium '96*, June 1996.

[Bukowski and Séquin, 1997] Richard Bukowski and Calro Séquin. Interactive simualtion of fire in virtual building environments. *Computer Graphics (SIGGRAPH 1997)*, 1997.

[Carmack et al., 1996] John Carmack, John Romero, and Michael Abrash. Quake, 1996. Computer Application (Game), screen shots courtesy of John Carmack, IDSoftware, http://www.idsoftware.com.

[Carpenter, 1984] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *Computer Graphics (SIGGRAPH 1984)*, pages 103–108, 1984.

[Christensen et al., 1998] T. Christensen, J. M. Knudsen, H. Stub, and H. Stub. Hvad er ild? *("what is fire?"). Illustreret Videnskab*, 4, 1998.

[Clausen, 1998] Jesper Clausen. Beregning af realistiske overflader. Master's Thesis, December 1998.

[Ebert and Parent, 1990] David S. Ebert and Richard E. Parent. Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques. *Computer Graphics (SIGGRAPH 1990)*, pages 357–366, August 1990.

[Ebert et al., 1998] David S. Ebert, F. Kenton Musgrave, Darwin Peachey, and Steven Worley. *Texturing and Modeling: A Procedural Approach.* AP Professional, second edition, July 1998.

[Elias, 1998] Hugo Elias. Models & natural phenomena. Misc. webpages, 1997-1998. http://freespace.virgin.net/hugo.elias/.

[Engell-Nielsen et al., 1994] Theo Engell-Nielsen, Thomas Albert, and Claus Beyer. Three dimensional user interfaces - a cognitive walk-through. Report written at Laboratory of Psychology, University of Copenhagen, 1994.

[Eysenck and Keane, 1990] Michael W. Eysenck and Mark T. Keane. *Cognitive psychology: a student's handbook.* Lawrence Erlbaum Associates Ltd., Hove, UK, 1990.

[Fire-fighting, 1999] Fire-fighting. "fire on the web"-page. Web-site, 1999. http://www.bfrl.nist.gov/info/fire.html.

[Foley et al., 1990] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addison-Wesley, Reading, Massachusetts, 2nd. edition, 1990.

[Forcade, 1995] Tim Forcade. *3D Studio IPAS Plug-In Reference.* New Riders Publishing, Indianapolis, Indiana, 1995.

[Foster, 1998] Nick Foster. *Errata for the "Modeling the Motion of a Hot, Turbulent Gas" article by Nick Foster & Dimitri Metaxas.* http://www.cis.upenn.edu-/~fostern/home.html, 1997/1998.

[Foster and Metaxas, 1997a] Nick Foster and Dimitri Metaxas. *Realistic Animation of Liquids.* http://www.cis.upenn.edu/~fostern/liquids.html, 1997.

[Foster and Metaxas, 1997b] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. *Computer Graphics (SIGGRAPH 1997)*, pages 181–188, August 1997.

[Founier et al., 1982] A. Founier, D. Fussel, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[Galea, 1997] Ed Galea. *Principles and Practice of Fire Modeling.* University of Greenwich, 1997.

[Galea, 1998] Ed Galea. Different computer animations. Published on the internet site of University of Greenwich, 1998. http://fseg.gre.ac.uk/.

[Gardner, 1985] Geoffrey Y. Gardner. Visual simulation of clouds. *Proceedings of SIGGRAPH*, 19(3):297–303, July 1985.

[Harlow and Welch, 1965] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8:2182–2189, 1965.

[Humphreys and Bruce, 1989] Glyn W. Humphreys and Vicki Bruce. *Visual Cognition: Computational, Experimental, and Neuropsychological Perspectives.* Lawrence Erlbaum Associates Ltd., Hove, UK, 1989.

[Jensen and Christensen, 1998] Henrik Wann Jensen and Per H. Christensen. Effecient simulation of light transmission in scenes with participating media using photon maps. *Computer Graphics (SIGGRAPH 1998)*, Proceedings:311–320, June 1998.

[Jones, 1993] Walter W. Jones. Modeling fires—the next generation of tools. *Web-published paper*, 1993. Group Leader, Fire Modeling and Application Group, Building and Fire Research Laboratory, National Insitute of Standards and Technology, Gaithersburg.

[Kajiya, 1982] James T. Kajiya. Ray tracing parametric patches. *Computer Graphics (SIGGRAPH 1982)*, pages 245–254, july 1982.

[Kajiya, 1983] James T. Kajiya. New technologies for raytracing procedurally defined objects. *Computer Graphics (SIGGRAPH 1983)*, pages 91–102, july 1983.

[Kajiya, 1989] James T. Kajiya. Rendering fur with three dimensional textures. *Proceedings of SIGGRAPH 1989*, 23(3):271–280, july 1989.

[Kajiya and von Herzen, 1984] James T. Kajiya and B. P. von Herzen. Raytracing volume densities. *Computer Graphics (SIGGRAPH 1984)*, pages 165–174, july 1984.

[Kincaid and Cheney, 1991] David Kincaid and Ward Cheney. *Numerical Analysis—Mathematics of Scientific Computing.* Brooks/Cole Publishing Company, international student edition edition, 1991.

[Kinetix, 1996] Autodesk / Kinetix. 3d studio max / elemental plugin. 3D graphics rendering application for Microsoft Windows, 1996. http://www.ktx.com/.

[Landau and Lifshitz, 1995] Lev Davidovich Landau and Evgenni Mikhailovich Lifshitz. *Fluid Mechanics.* Pergamon Press, 2nd edition, 1995.

[Laur and Hanrahan, 1991] David Laur and Pat Hanrahan. Hierarchial splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics*, 25(4):285–288, July 1991.

[Lenoble, 1985] J. Lenoble. *Radiative Transfer in Scattering and Absorbing Atmospheres.* A. Deepak Publishing, Hampton, Virginia, 1985.

[Lewis, 1989] J. P. Lewis. Algorithms for solid noise synthesis. *Proceedings for SIGGRAPH*, 23(3):263–270, July 1989. Also published as ACM Computer Graphics.

[Lide, 1992] David R. Lide. *Handbook of Chemistry and Physics.* CRC Press, 73 edition, 1992.

[Mandelbrot and van Ness, 1968] B. Mandelbrot and J. W. van Ness. Fractional brownian motion, fractional noises and applications. *SIAM Review*, 10(4):321–340, 1968.

[Mell et al., 1996] William E. Mell, Kevin B. McGrattan, and Howard R. Baum. Numerical simulation of combustion in fire plumes. *26th Symposion (International) on Combustion*, pages 1523–1530, 1996.

[Mogensen, 1986] Torben Æ. Mogensen. The application of partial evaluation to raytracing. *Unpublished*, 1986.

[Murray and van Ryper, 1994] James D. Murray and William van Ryper. *Encyclopedia of Graphics File Formats.* O'Reilly & Associates, Inc., Sebastopol, CA 95472, United States of America, first edition, 1994.

[Musgrave, 1993] Ken Musgrave. Fire renderings. Only images are available here, 1983–1993. http://www.seas.gwu.edu/faculty/musgrave/fire.html.

[Newtek, 1999] Newtek. Hypervoxel 2.0. Application plugin for the commercial visualisation tool Lightwave, March 1999. http://www.newtek.com/.

[Paramount, 1982] Paramount. Star trek: The wrath of khan. movie, june 1982.

[Parbo, 1986] Henrik Parbo. *Bag den farvede virkelighed* ("Behind the colours of reality"). systime, 1986.

[Perlin, 1985] Ken Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH 1985)*, 19(3):287–296, 1985.

[Perlin and Hoffert, 1989] Ken Perlin and Eric Hoffert. Hypertexture. *Computer Graphics (SIGGRAPH 1989*, pages 253–262, july 1989.

[Press et al., 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[Reeves, 1983] William T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *Computer Graphics (SIGGRAPH 1983)*, 17(3):359–376, 1983.

[Rushmeier and Torrance, 1987] Holly E. Rushmeier and Kenneth E. Torrance. The zonal method for calculating light intensities in the presence of a participating medium. *Computer Graphics (SIGGRAPH 1987)*, 21(4):293–302, 1987.

[Schreiber et al., 1994] Schreiber, Yost Group, and Autodesk. 3D Studio ver. 4 / Flames IPAS from Schreiber Instruments. 3D graphics rendering application, 1994.

[Schröder and Hanrahan, 1993] Peter Schröder and Pat Hanrahan. A closed form expression for the form factor between two polygons. *Tech. Report, Department of Computer Science, Princeton University*, CS–404–93, January 1993.

[Sims, 1990] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics (SIGGRAPH 1990)*, 24(4):405–413, 1990.

[Stam, 1991] Jos Stam. *A Multi-Scale Stochastic Model for Computer Graphics.* Master Thesis, Department of Computer Science, University of Toronto, 1991.

[Stam, 1995] Jos Stam. *Multi-Scale Stochastic Modeling of Complex Natural Phenomena.* Ph.D. Thesis, Department of Computer Science, University of Toronto, 1995.

[Stam and Fiume, 1993] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH 1993 Proceedings)*, volume 27, pages 369–376, August 1993.

[Tritton, 1988] D. J. Tritton. *Physical Fluid Dynamics.* Oxford University Press, 2nd edition, 1977, 1988.

[Varlin, 1973] Charles H. Varlin, editor. *Fire Protection Manual.* Gulf Publishing Company, Book Division, Houston, Texas, 2nd edition, 1964, 1973.

[Wallace et al., 1989] J. R. Wallace, K. E. Elmquist, and E. A. Hanes. A ray tracing algorithm for progressive radiosity. *ACM Computer Graphics (SIGGRAPH 1989)*, 23(3):315–324, 1989.

[Watkins et al., 1992] Christopher D. Watkins, Stephen B. Coy, and Mark Finlay. *Photorealism and Ray Tracing in C.* M&T Books, Inc., 1992.

[Watt and Watt, 1992] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques Theory and Practice*, chapter 15. Addsion-Wesley, Wokingham, England, 1992.

[Wyvill et al., 1986] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[Yaglom, 1986] A. M. Yaglom. *Correlation Theory of Stationary and Related Random Functions I. Basic Results.* Springer Verlag, 1986.

Index