Interactive visualisation techniques for large time-dependent data sets

Interactive visualisation techniques for large time-dependent data sets

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 12 juni 2007 om 15.00 uur

door

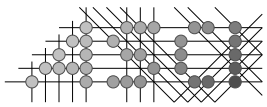
Benjamin VROLIJK

informatica ingenieur
geboren te Katwijk.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. F.W. Jansen

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof. dr. ir. F.W. Jansen, Technische Universiteit Delft, promotor
Ir. F.H. Post, Technische Universiteit Delft, toegevoegd promotor
Prof. dr. ir. C. Vuik, Technische Universiteit Delft
Prof. dr. M. Gross, Eidgenössische Technische Hochschule Zürich
Prof. dr. F.J. Peters, Universiteit Leiden
Prof. dr. ir. H.J. Sips, Technische Universiteit Delft
Dr. ir. B.J. Boersma, Technische Universiteit Delft



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 147.

Netherlands Organisation for Scientific Research

This project was supported by the Netherlands Organisation for Scientific Research (NWO) under the Computational Science research programme through grant number 635.000.004.

ISBN 978-90-8559-293-8

Preface

The research described in this thesis was performed at the Computer Graphics and CAD/CAM research group of the Delft University of Technology. This research was part of a larger project, supported by the Netherlands Organisation for Scientific Research (NWO) on the NWO-EW Computational Science Programme "Direct Numerical Simulation of Oil/Water Mixtures Using Front Capturing Techniques".

The project was a cooperation between the areas of Fluid Mechanics, Numerical Analysis and Computer Graphics. The main goal of this part of the research project was to visualise phase fronts in multi-phase flows and to study their evolution in time and, more in general, to develop methods for efficient handling of large, time-dependent data sets.

A lot of people are involved in making a project like this happen. Therefore, I would like to thank all the people who have made it possible for me to do this project and write this thesis.

First of all, Frits, my supervisor. Thank you for your support, your many fruitful ideas and the endless, always inspiring, discussions.

Erik, my promotor, for giving me the opportunity to do this research at the Computer Graphics group and for the quick and accurate reviewing of my thesis.

My co-researchers on the same project: Bendiks Jan Boersma for initiating and supervising the entire project and Sander van der Pijl and Emil Coyajee, the other PhD students, for the pleasant cooperation.

I would like to thank Helwig Hauser, Helmut Doleisch and Bob Laramee from VRVis Research Center in Vienna, for our joint work on the state-of-the-art report.

All the people of the Computer Graphics group for creating such a pleasant working atmosphere. The PhD students and postdocs: Alex, Charl, Eelco, Eric, Gerwin,

Lingxiao, Michal, Paul, Rafa, Rick, Wouter. All the Master's students who have come and gone, and those who didn't want to go.

Yang Yang, thank you for the valuable work you have done, both during your research project and your Master's project.

Ruud and Bart for the technical support and Toos for the administrative and organisational support. Where would we be without you?

I would like to thank my family and my parents in particular for their never-ending love and support.

Finally, Irene, the love of my life, for believing in me, supporting me in everything and simply for being part of my life. Thank you!

Benjamin Vrolijk

Contents

CHAPTER 1

Introduction

1.1 Visualisation

Visualisation is a new and exciting topic within the field of computer graphics. It is the science (or even the "art") of turning large amounts of data into an image or a visual representation that provides insight into the structure and properties of the data.

Visualisation utilises the powerful capabilities of the human visual system. The easiest way for us to process large amounts of information is through the visual system. Therefore, it is much easier for us to interpret a weather map than it is to interpret a table of locations and temperatures. Figure 1.1 on the following page illustrates this example.

Depending on the type and source of the data, several subfields of visualisation can be distinguished. Two main subfields are information visualisation and scientific (or data) visualisation. Information visualisation is the subfield that is mainly concerned with information from databases, such as tabular and structural data. Scientific visualisation, on the other hand, is about physical data. This field by itself can also be subdivided. For example, medical visualisation is about patient data that has been acquired using MRI and CT scanners, or similar. As another example, flow visualisation is visualisation of data that results from measurements and simulations

Weather station	Temperature (°C)
Terschelling	26.1
Den Helder	25.7
Schiphol	23.6
Rotterdam	23.6
Vlissingen	25.2
Leeuwarden	28.7
Eelde	30.2
Twenthe	30.5
Lelystad	20.6
De Bilt	23.4
Eindhoven	22.7
Maastricht	23.6

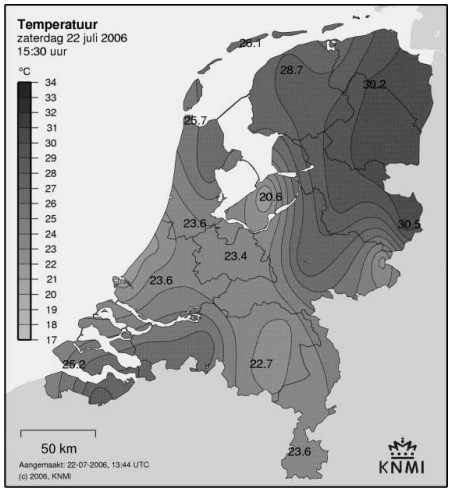


Figure 1.1: A temperature table and accompanying temperature map for The Netherlands. See also colour Figure C.1. (Source: KNMI)

of fluid flows. Each of these subfields has its own set of typical visualisation algorithms, which follow from the type of data that is used.

A very nice and well-known example of information visualisation is the map of Napoleon's invasion of Russia in 1812, by Charles Minard (Figure 1.2 on the next page). It shows the size of the French army during the march into Russia (light) and the retreat (dark). The width of the bars depicts the number of troops. Other types of information that can be read from this graph include dates, places and temperatures.

Medical visualisation is concerned with transforming the data that has been produced by MRI or CT scanners into insightful images. These scans often result in stacks of two-dimensional greyscale images, which can be seen as slices through a human body, for example. First, these stacks have to be reconstructed to a three-dimensional volume. Then, visualisation algorithms can be used, for example, to show (the surface of) bones, or to visualise soft tissue on arbitrary slices (see Figure 1.3 on page 4).

Flow visualisation data originates from either measurements or computer simulations of flows. You can think of the airflow around a plane, (see Figure 1.4 on page 5,) the flow of oil through a pipeline, but also the flow of a river. This type of data often consists of multiple quantities, such as pressure, temperature and velocity, as opposed to medical data, which normally consists of a single value. Inherent in flows is that they change over time. Often, the measurements or simulations will be done on discrete time steps over a period of time, resulting in huge, time-dependent data sets. So, whereas medical data can be seen as a stack of images, time-dependent flow

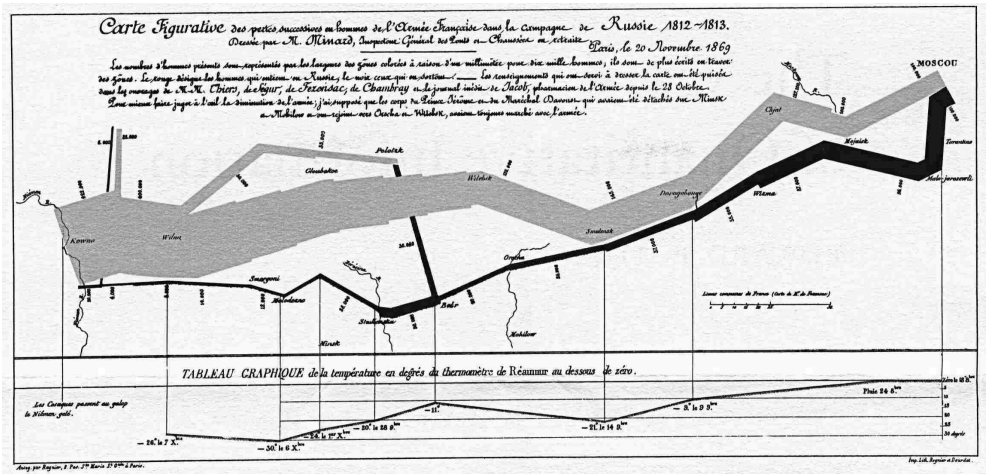


Figure 1.2: Minard's map of Napoleon's march against Russia [92].

data can be seen as a "stack" of 3D volumes.

In all cases, the purpose of the visualisation is to present to the user an image (or a movie in the case of time-dependent data) that the user can interpret more easily than the raw data. Even the simple example in Figure 1.1 demonstrates this. Imagine the amount of data that lies underneath Figure 1.4 and how utterly useless a tabular representation would be in this case.

1.2 Large data handling

One of the main research problems in scientific visualisation today is how to deal with the enormous amounts of data and information that are available [27]. These data could be generated, for example, by simulations on supercomputers but also by acquisition devices such as MRI scanners.

High-resolution time-varying data sets, containing several scalar, vector and/or tensor fields are very common. In Computational Fluid Dynamics, for example, simulations can easily produce several gigabytes or even terabytes of data.

Unfortunately, this is not a transitory problem: it will not be solved by waiting for the new generation of computers. As computing power increases, data set sizes increase even more. Supercomputers are getting bigger and faster, processor speed and memory size are continually increasing. Because of this, i.e. because it is possible, simulations that are done on supercomputers are becoming more and more accurate.

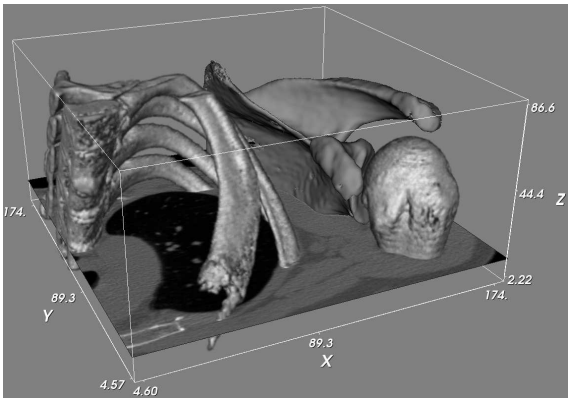


Figure 1.3: An example of medical visualisation. A surface rendering of bones in the shoulder, together with a slice showing the soft tissue. (Source: C.P. Botha)

As a result, the average size of data sets is growing faster than the capacities of (personal) computers.

It is obvious that special techniques are needed to tackle this large data handling problem. New techniques will have to be developed for handling the data, for storing the data sets on disk and in memory, for compressing the data, and for visualisation.

We are facing the problem of large data handling in general, and of interactive visualisation of time-varying data in particular.

There are a number of approaches for solving the large data handling problem. In the following these approaches will be categorised into three classes.

Data reduction

The first class of approaches tries to reduce the amount of data that has to be stored and processed as much as possible.

Compression is a data reduction approach that can be described as statistics-based data reduction. Numerous data compression algorithms exist, many originating from the field of image processing. Compression techniques can be subdivided into lossless and lossy techniques, depending on whether or not the original data can be reconstructed perfectly from the compressed data.

Often, a quantisation step is involved to further reduce the amount of data; this will also inevitably make the compression lossy. Techniques from the field of image processing have naturally been developed for compressing 2D images. However, many of

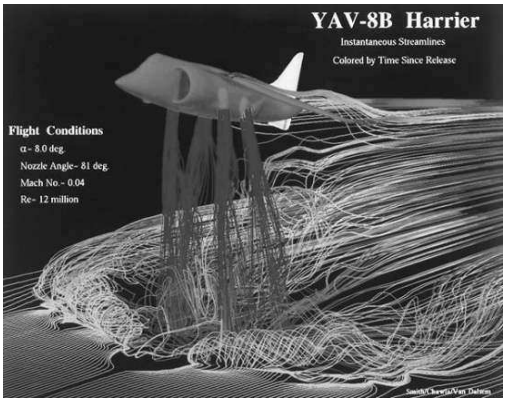


Figure 1.4: An example of flow visualisation. The flow around a Harrier aircraft, shown using streamlines. The colour indicates the time since "release". See also colour Figure C.2. (Source: aerospaceweb.org)

these techniques can easily be extended to 3D data sets and even 4D (time-dependent) data sets.

Feature extraction can be described as content-based data reduction. Features are those objects, structures or phenomena in the data, that are of interest. The definition of a feature naturally depends on the application and even on the researcher. But with a good feature definition, feature extraction will reduce the data to that which is of interest. This can easily give a reduction of a factor 1 000 or more. Features abstract the data. They can be represented very compactly and described as individual objects. For time-dependent data sets, feature tracking can be performed to determine the temporal evolution of each of the features and even distinguish temporal events in the evolution. This makes feature-based visualisation a very powerful approach to time-dependent data handling.

Because of the abstraction of the data, feature extraction is an approach that is more or less orthogonal to the other approaches. The process of extraction and abstraction will result in the description of objects with attributes such as size or volume. The purpose of this approach is to reduce the original data to these more informative objects. The original data will no longer be needed, because the objects are used for further processing and visualisation.

Special data structures

The second class of approaches focuses on specific data structures to ease the task of large data handling.

Fast-access data structures can be used, that are designed to perform certain visualisation tasks efficiently. For example, special data structures exist for efficient isosurface extraction or fast volume rendering. Data structures such as these will normally have to be created in preprocessing. This takes time, but upon using the data structure, the performance will be significantly better than with the raw data. The drawback is that most of these data structures have been designed to perform a specific visualisation task, and will therefore be limited to this task. Combining different visualisations of the same data set will either be very slow, or require several special data structures.

Multi-resolution data structures provide another solution to the handling of large data sets. This approach often involves increasing the total amount of data by storing the data set at several levels of resolution. Interactive visualisation is possible by using a low resolution version of the data set, if necessary. When time and memory space allow it, a higher resolution version of the data can be loaded.

Disk and I/O techniques

The third class of approaches is concerned with managing and optimising the transfer of data, either from disk to main memory or across a network connection.

Out-of-core or external memory techniques try to overcome the bottleneck that is the I/O communication between secondary and main memory. This can be done by minimising the amount of data that has to be transferred and optimising the coherence of disk access. I/O-optimised and cache-friendly data structures are required for this.

A large amount of work has been done on external memory algorithms. For an overview of out-of-core algorithms for computer graphics and visualisation, we refer to the survey by Silva et al. [82].

Streaming is a technique where data is transferred (possibly across a network) in a stream of packets which are interpreted as they arrive. Streaming is a well-known and commonly used technique on the Internet to transmit multimedia files. Streaming audio and video can be played back as it is being downloaded. Broadcast radio and television signals are probably the most familiar examples of (inherently) streaming data.

However, for visualisation applications, streaming of 3D or 4D data sets is required. Algorithms have to be adapted to work with one-dimensional data streams. Not much work has been presented in this area. Only, more generally, in the area of computer graphics applications, work has been done on streaming of 3D scenes [74], or streaming of meshes [70, 24].

Slightly related to streaming techniques are techniques that have been designed to run on the Graphics Processing Unit or GPU. The purpose of these techniques is not

primarily to solve the large data handling problem, therefore they have not played a substantial role in this research. However, the techniques could be categorised in this approach, because they deal with the data transfer from main memory to the graphics card. As such, some of the GPU-based techniques could be classified as streaming algorithms.

Due to the rapid development of graphics hardware in recent years, there has been a huge increase in research on hardware-accelerated or GPU-based algorithms [63, 106, 50]. Existing visualisation algorithms have been adapted and new algorithms have been designed to work on the programmable graphics hardware. To this end, the so-called vertex and/or fragment programs are loaded into the GPU and as the data is streamed to the graphics card, the corresponding programs are run for each vertex or pixel. For example, Pascucci presented an algorithm for hardware-accelerated isosurface computation [50], in which the CPU is concerned with streaming tetrahedra to the graphics card, and the vertex program on the GPU converts each tetrahedron into a single quad of the isosurface.

1.3 Project

The research described in this thesis is part of a larger research project about multi-phase flows. These are flows consisting of two or more fluids that do not mix. You can think of air bubbles or oil droplets in water. Such "bubbly flows" are common in nature, but also have important applications in industry, for example in chemical reactors or fuel injectors.

A multi-phase flow is characterised by the sharp boundaries between two fluids, the so-called phase fronts. Because there is no mixture, the transition from one fluid (or phase) to another is very sharp and coincides with large jumps in physical quantities such as density and viscosity.

One of the goals of this research project is to accurately simulate multi-phase flows, using Computational Fluid Dynamics or CFD. This involves solving the governing laws of fluid dynamics numerically. These laws are described by a set of partial differential equations, known as the Navier-Stokes equations. To solve these equations on a computer, the continuous fluid has to be discretised, for example by discretising the spatial domain into a mesh or grid.

However, on discretised grids, the sharp transitions within multi-phase flows cannot be represented very well. Because of the finite grid size, the transitions will be smeared out over at least the size of one grid cell. Special precautions have to be taken to minimise this smearing effect, for example by using an implicit or explicit surface representation of the phase front.

Other difficulties in the simulation arise from the fact that each of the phases must be solved separately, and that the phases form each other's boundary conditions. The phases influence each other and the (moving) boundary in between.

Part of the goal to accurately simulate multi-phase flows is to develop efficient methods for the Direct Numerical Simulation (DNS) of these flows.

Another goal of the project is to study the evolution of the phase fronts, i.e. to study the development of the surfaces over time and to understand how they change and interact with each other. Imagine the air bubbles rising in water. What will happen if two bubbles coalesce? How will the surfaces merge?

In order to study the evolving fronts, methods are needed for detecting and extracting them in the first place, and subsequently for tracking the phase fronts over time.

Besides the physical (simulation) and numerical aspects, a third aspect of the research in this project is the visualisation. Visualisation of the fluid flow and of the phase fronts in particular is required to gain insight into both the simulation and the flow itself. The development of techniques for the interactive visualisation of the evolving phase front is another one of the research goals.

The focus of my research — the visualisation part of this project — is on these last two goals: the detection, extraction and tracking of the phase front, combined with the efficient visualisation of the evolution of the phase front.

The process from flow simulation to visualisation of the phase front can be split into a number of steps. First, the simulation runs and results in a large "raw" data set on disk. Then, the surface interface or phase front has to be detected and extracted from this data set. Finally, the surface can be visualised using more or less specialised techniques.

Initially, in this research project, the precise methods and algorithms that were to be used, had not yet been specified. Therefore, we started exploring feature extraction and tracking techniques, in order to be able to detect and extract the phase front from the raw data set.

However, after about a year into this project, it became apparent that the numerical method that was to be used for the simulation already included an implicit surface representation of the phase front. This surface representation could be stored with or within the raw data set and could then easily be extracted from the data as an isosurface. Therefore, we abandoned our research into feature-based techniques at that time.

Independent of the method that is used, the last step in the process is the visualisation of the phase front. Assuming that the phase front has either been extracted explicitly using feature extraction, or has been represented and stored separately during the simulation, the visualisation of the phase front will be straightforward.

Therefore, the focus of my research was directed towards efficient techniques for interactive isosurfacing from very large data sets. Two important aspects are involved here. The first is interactive isosurfacing. We need a technique that can perform isosurface extraction from time-dependent data sets, and we need this technique to be very fast in order to be able to browse interactively through the data set. The second aspect is that we have to work with huge data sets. These data sets will consist of very large grids and contain hundreds or even thousands of time steps. The size on disk will be many gigabytes, or even terabytes. In relation to a personal computer's main memory of maybe one or two gigabytes, it is clear that this poses a huge problem for interactive handling of the data.

At first, the goal of my research was to look into methods for visualisation of isosurfaces from very large, time-dependent data sets. After that, and more in general, the goal of my research became to investigate methods for efficient handling of large, time-dependent data sets for visualisation purposes. The time dimension is an important aspect, because on the one hand, this adds another order of magnitude to the size of the data sets, but on the other hand, in order to investigate time-dependent data, interactivity is of the essence.

1.4 Overview of this thesis

We have reviewed the state-of-the-art techniques and algorithms in the field of flow visualisation. The results have been presented on the annual Eurographics conference in Saarbrücken in 2002 and have been published in two separate articles in Computer Graphics Forum [56, 34]. Chapter 2 will present our overview of feature-based flow visualisation techniques. Feature extraction techniques for several important types of flow features will be discussed, as well as algorithms for feature tracking and event detection. The Chapter will conclude with a description of feature visualisation techniques.

Chapter 3 will introduce a number of techniques to tackle the problem of very large data handling, following the classification from Section 1.2. Techniques using (a combination of) compression, multi-resolution and/or fast-access data structures will be presented.

In Chapters 4 and 5, one specific fast-access data structure will be discussed, that has been the main focus of this project. We have developed algorithms for fast isosurfacing from time-dependent data sets, achieving interactive frame rates for grids up to 256^3 data points per time step. We have used specialised data structures to make use of temporal coherence in the data and to provide fast isosurface cell selection. We have developed an "incremental" search algorithm fitting these data structures and likewise making use of the temporal coherence in the data.

We have combined the fast extraction algorithm with specialised out-of-core techniques. We have developed a time window technique to control the amount of data that is in memory while still maintaining interactive frame rates in order to support the handling of very large size data sets on a normal personal computer. This work has been presented at the Spring Conference on Computer Graphics in Budmerice, Slovakia in 2004 [96] (Chapter 4), and has been published in Computers & Graphics in 2006 [97] (Chapter 5).

We have further investigated other techniques that support large data handling for visualisation, such as multi-resolution techniques, which are not limited to isosurfacing, but allow for a wider range of visualisation algorithms. In Chapter 6 a multi-resolution technique will be described, that has been extended for time-dependent data sets and space-time navigation.

Finally, Chapter 7 will give the conclusions of my thesis and propose directions for future research.

CHAPTER 2

The state of the art in flow visualisation: feature extraction and tracking

Frits H. Post, Benjamin Vrolijk, Helwig Hauser, Robert S. Laramee and Helmut Doleisch

Computer Graphics Forum
vol. 22, no. 4, December 2003, pages 775–792, ISSN 0167-7055.

Abstract

Flow visualisation is an attractive topic in data visualisation, offering great challenges for research. Very large data sets must be processed, consisting of multivariate data at large numbers of grid points, often arranged in many time steps. Recently, the steadily increasing performance of computers again has become a driving force for new advances in flow visualisation, especially in techniques based on texturing, feature extraction, vector field clustering, and topology extraction.

In this article we present the state of the art in feature-based flow visualisation techniques. We will present numerous feature extraction techniques, categorised according to the type of feature. Next, feature tracking and event detection algorithms are discussed, for studying the evolution of features in time-dependent data sets. Finally, various visualisation techniques are demonstrated.

2.1 Introduction

Flow visualisation is one of the traditional subfields of data visualisation, covering a rich variety of applications, ranging from automotive, aerospace, and turbomachinery design, to weather simulation and meteorology, climate modelling, and medical applications, with many different research and engineering goals and user types. Consequently, the spectrum of flow visualisation techniques is very rich, spanning multiple dimensions of technical aspects, such as 2D and 3D techniques, and techniques for steady and time-dependent data.

In this article we present the state of the art in flow visualisation techniques. These techniques can be categorised into four groups:

- *Direct flow visualisation*: The data is directly visualised, without much preprocessing, for example by colour-coding or drawing arrows. These techniques are also called *global techniques*, as they are usually applied to an entire domain, or a large part of it.

- *Texture-based flow visualisation*: Texture-based techniques apply the directional structure of a flow field to random textures. These are mainly used for visualising flow in two dimensions or on surfaces. The results are comparable to the experimental techniques like wind tunnel surface oil flows. This group has some characteristics of the previous and the next approaches.

- *Geometric flow visualisation*: Geometric objects are first extracted from the data, and used for visualisation. Examples are streamlines, stream surfaces, time surfaces, or flow volumes. These geometric objects are directly related to the data. The results of these techniques can be compared to experimental results such as dye advection or smoke injection into the flow.

- *Feature-based flow visualisation*: The last approach lifts the visualisation to a higher level of abstraction, by extracting physically meaningful patterns from the data sets. The visualisation shows only those parts that are of interest to the researcher, the *features*. Both the definition of what is interesting, and the way these features are extracted and visualised are dependent on the data set, the application, and the research problem.

The approaches are not entirely distinct. For example, the second and third approaches can be combined into *dense flow visualisation*.

In this article, we survey the last approach, feature-based flow visualisation.

Features are phenomena, structures or objects in a data set, that are of interest for a certain research or engineering problem. Examples of features in flow data sets

are shock waves, vortices, boundary layers, recirculation zones, and attachment and separation lines.

There are a number of factors motivating the feature-based approach to visualisation. First, by extracting only the interesting parts, and ignoring the rest, we can increase the information content. Furthermore, by abstracting from the original data, the researcher is able to focus more on the relevant physical phenomena, which is better related to his conceptual framework. A large data reduction can be achieved (in the order of 1000 times), but because the reduction is content-based, no (important) information is lost. So far, this is one of the few approaches that is truly scalable to very large time-dependent data sets. Finally, the objects or phenomena extracted can be simplified and described quantitatively. This makes the visualisation easy, using simple geometries or parametric icons. Also, quantification facilitates further research, comparison and time tracking.

The paper is structured as follows: in the next Section, we will discuss some fundamentals for flow visualisation, which are necessary for understanding the rest of the paper. In Section 2.3 an introduction to feature extraction is given, with a categorisation of the general approaches to feature extraction. In Section 2.4 feature extraction techniques are discussed, for several different types of features. Section 2.5 discusses feature tracking and event detection, that is, the study of the evolution of features in time-dependent data sets. Section 2.6 presents different iconic representations of features and the visualisation of features and events. Finally, in Section 2.7 some conclusions and further prospects are presented.

2.2 Flow visualisation fundamentals

For a proper understanding of the rest of the article, it is necessary to discuss a number of fundamentals for flow visualisation, mainly from vector algebra.

2.2.1 Gradients

In three dimensions, a scalar p has three partial derivatives. The partial derivative of p with respect to $\mathbf{x}$ is $\frac{\partial p}{\partial \mathbf{x}}$. The gradient of a scalar field is the vector of its partial derivatives:

$$\text{grad}p = \nabla p = [\frac{\partial p}{\partial \mathbf{x}} \quad \frac{\partial p}{\partial \mathbf{y}} \quad \frac{\partial p}{\partial \mathbf{z}}]. \tag{2.1}$$

The gradient of a vector field $\mathbf{v}$ is found by applying the gradient operator to each of the components $[u \;\; v \;\; w]$ of the vector field. This results in a 3×3 matrix, called the

Jacobian of the vector field, or the matrix of its first derivatives:

$$\nabla \mathbf{v} = \begin{bmatrix} \frac{\partial u}{\partial \mathbf{x}} & \frac{\partial u}{\partial \mathbf{y}} & \frac{\partial u}{\partial \mathbf{z}} \\ \frac{\partial v}{\partial \mathbf{x}} & \frac{\partial v}{\partial \mathbf{y}} & \frac{\partial v}{\partial \mathbf{z}} \\ \frac{\partial w}{\partial \mathbf{x}} & \frac{\partial w}{\partial \mathbf{y}} & \frac{\partial w}{\partial \mathbf{z}} \end{bmatrix} \tag{2.2}$$

This matrix can be used to compute a number of derived fields, such as the divergence, curl, helicity, acceleration, and curvature. The curl of a *velocity* field is called the *vorticity*. This derived vector field indicates how much the flow locally rotates and the axis of rotation. These quantities are all used in different feature extraction techniques, which will be discussed later. The exact definitions can be found elsewhere [36, 66]. For the understanding of this article, it is sufficient to know that the Jacobian, or gradient matrix, is an important quantity in flow visualisation in general and in feature extraction in particular.

2.2.2 Eigenanalysis

Another indispensable mathematical technique is eigenanalysis. An *eigenvalue* of a 3×3 matrix M is a (possibly complex) scalar λ which solves the eigenvector equation: $M\mathbf{x} = \lambda\mathbf{x}$. The corresponding non-zero vector $\mathbf{x}$ is called an *eigenvector* of M. The eigenvectors and eigenvalues of a Jacobian matrix indicate the direction of tangent curves of the flow, which are used, for example to determine the vector field topology, see Section 2.3.2.

2.2.3 Attribute calculation

As a part of the feature extraction process, characteristic attributes of the features have to be calculated. One conceptually simple and space efficient technique, is the computation of an ellipsoid fitting. An ellipsoid can give a first-order estimation of the orientation of an object. The axes can be scaled to give an exact representation of the size or volume of the object. Furthermore, an ellipsoid is a very simple icon to visualise. The computation of an ellipsoid fitting involves eigenanalysis of the *covariance matrix* of the object's grid points. For a detailed description, see Haber and McNabb [18], Silver et al. [85] and De Leeuw [36].

Another technique that can be used for attribute calculation of features is *centre line extraction*. As an example, a skeleton, or Medial Axis Transform, reduces an object to a single centre line, or graph, while preserving the original topology of the object. Using this graph, an icon can be constructed from cylinders and hemispheres, to construct an approximation of the original shape of the object [58]. This is a useful representation, especially when the topology is an important characteristic of the features.

2.3 Feature extraction approaches

Feature-based flow visualisation is an approach for visualising the flow data at a high level of abstraction. The flow data is described by features, which represent the interesting objects or structures in the data. The original data set is then no longer needed. Because often, only a small percentage of the data is of interest, and the features can be described very compactly, an enormous data reduction can be achieved. This makes it possible to visualise even very large data sets interactively.

The first step in feature-based visualisation is *feature extraction*. The goal of feature extraction is determining, quantifying and describing the features in a data set.

A feature can be loosely defined as any object, structure or region that is of relevance to a particular research problem. In each application, in each data set and for each researcher, a different feature definition could be used. Common examples in fluid dynamics are vortices, shock waves, separation and attachment lines, recirculation zones and boundary layers. In the next Section a number of feature-specific detection techniques will be discussed. Although most feature detection techniques are specific for a particular type of feature, in general the techniques can be divided into three approaches: based on image processing, on topological analysis, and on physical characteristics.

2.3.1 Image Processing

Image processing techniques were originally developed for analysis of 2D and 3D image data, usually represented as scalar (greyscale) values on a regular rectangular grid. The problem of analysing a numerical data set, represented on a grid, is similar to analysing an image data set. Therefore, basic image processing techniques can be used for feature extraction from scientific data. A feature may be distinguished by a typical range of data values, just as different tissue types are segmented from medical images. Edges or boundaries of objects are found by detecting sharp changes in the data values, marked by high gradient magnitudes. Thus, basic image segmentation techniques, such as thresholding, region growing, and edge detection can be used for feature detection. Also, objects may be quantitatively described using techniques such as skeletonisation or principal component analysis. However, a problem is, that in computational fluid dynamics simulations, often grid types are used such as structured curvilinear grids, or unstructured tetrahedral grids. Many techniques from image processing cannot be easily adapted for use with such grids. Furthermore, many digital filtering techniques are defined only for scalar data. Adaptation to vector fields is not always straightforward.

2.3.2 Vector Field Topology

A second approach to feature extraction is the topological analysis of 2D linear vector fields, as introduced by Helman and Hesselink [20, 21], which is based on detection and classification of critical points.

The critical points of a vector field are those points where the vector magnitude is zero. The flow in the neighbourhood of critical points is characterised by eigenanalysis of the velocity gradient tensor, or Jacobian of the vector field. The eigenvalues of the Jacobian can be used to classify the critical points as attracting or repelling node or focus, as saddle point, or centre. (See Figure 2.1.) The eigenvectors indicate the directions in which the flow approaches or leaves the critical point. These directions can be used to compute tangent curves of the flow near the critical points. Using this information, a schematic visualisation of the vector field can be generated. (See Figure 2.7 on page 28.) Helman and Hesselink have also extended their algorithm to 2D time-dependent and to 3D flows.

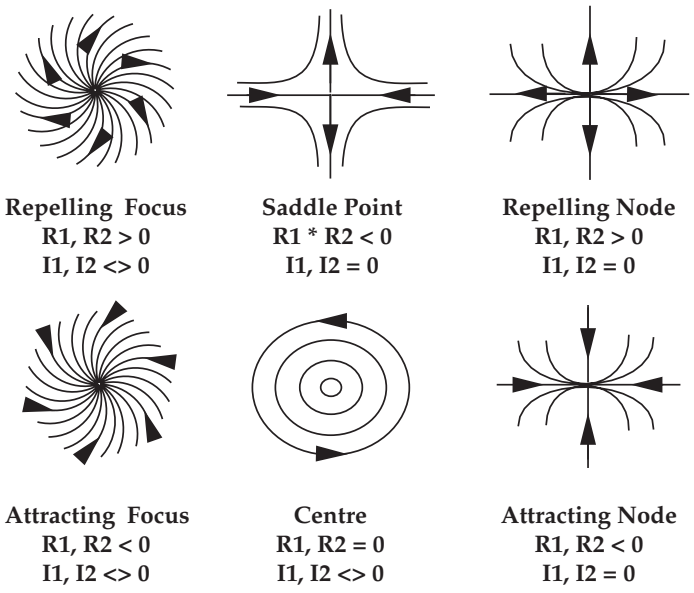


Figure 2.1: Vector field topology: critical points classified by the eigenvalues of the Jacobian [20].

Tricoche et al. recently presented a topology-based method for visualising time-dependent 2D vector fields [91]. They perform time tracking of critical points and closed streamlines by temporal interpolation. They are able to find and characterise

topological events or structural changes (*bifurcations*), such as the pairwise annihilation or creation of a saddle point and an attracting or repelling node.

Scheuermann et al. presented an algorithm for visualising nonlinear vector field topology [76], because other known algorithms are all based on piecewise linear or bilinear interpolation, which destroys the topology in case of nonlinear behaviour. Their algorithm makes use of Clifford algebra for computing polynomial approximations in areas with nonlinear local behaviour, especially higher-order singularities.

De Leeuw and Van Liere presented a technique for visualising flow structures using multilevel flow topology [38]. In high-resolution data sets of turbulent flows, the huge number of critical points can easily clutter a flow topology image. The algorithm presented attempts to solve this problem by removing small-scale structures from the topology. This is achieved by applying a pair distance filter which removes pairs of critical points, that are near each other. This removes small topological structures such as vortices, but does not affect the global topological structure. The threshold distance, which determines which critical points are removed, can be adapted, making it possible to visualise the structure at different levels of detail at different zoom levels.

Tricoche et al. also perform topology simplification in 2D vector fields [90]; they simplify not only the topology, but also preserve the underlying vector field, thereby making it possible to use standard flow visualisation methods, such as streamlines or LIC, after the simplification. The basic principle of removing pairs of critical points is similar to the technique of De Leeuw and Van Liere [38], but in this algorithm the vector field surrounding the critical points is slightly modified, in such a way that both critical points disappear.

2.3.3 Physical characteristics

The third approach is feature extraction based on physical characteristics. Often, features can be detected by characteristic patterns in, or properties of, physical quantities, for example by low pressure, high temperature, or swirling flow. These properties often follow directly from the feature definitions used. Most of the feature extraction techniques discussed in Section 2.4 are based on this approach, sometimes in combination with topological analysis or image processing techniques.

2.3.4 Selective Visualisation

A generic approach to feature extraction is Selective Visualisation, which is described by Van Walsum [98]. The feature extraction process is divided into four steps (see Figure 2.2 on the facing page).

The first step is the *selection* step. In principle, any selection technique can be

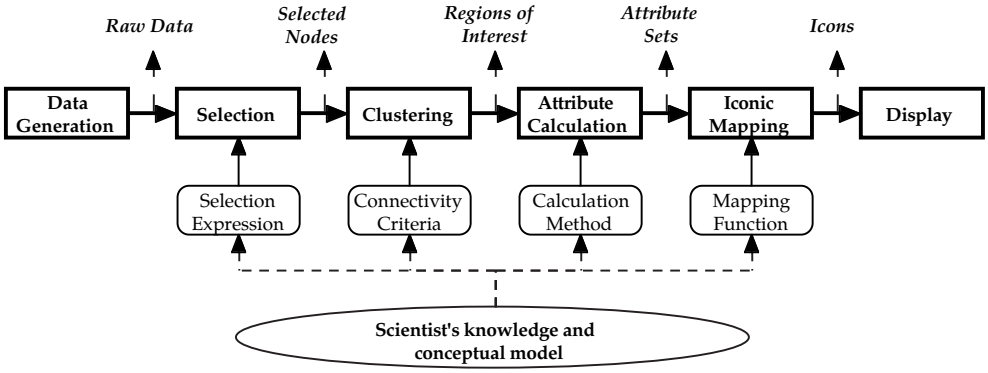


Figure 2.2: The feature extraction pipeline [62].

used, that results in a binary segmentation of the original data set. A very simple segmentation is obtained by thresholding of the original or derived data values; also, multiple thresholds can be combined. The data set resulting from the selection step is a binary data set with the same dimensions as the original data set. The binary values in this data set denote whether or not the corresponding points in the original data set are selected. The next step in the feature extraction process is the *clustering* step, in which all points that have been selected are clustered into coherent regions. In the next step, the *attribute calculation* step, these regions are quantified. Attributes of the regions are calculated, such as position, volume and orientation. We now speak of objects, or features, with a number of attributes, instead of clusters of points. Once we have determined these quantified objects, we don't need the original data anymore. With this, we may accomplish a data reduction factor of 1000 or more. In the fourth and final step, *iconic mapping*, the calculated attributes are mapped onto the parameters of certain parametric icons, which are easy to visualise, such as ellipsoids.

2.4 Feature extraction techniques

In this Section, a number of feature extraction techniques will be discussed that have been specifically designed for certain types of features. These techniques are often based on physical or mathematical (topological) properties of the flow. Features that often occur in flows are vortices, shock waves and separation and attachment lines.

2.4.1 Vortex extraction

Features of great importance in flow data sets, both in theoretical and in practical research, are *vortices*. (See Figure 2.3.) In some cases, vortices (turbulence) have to be impelled, for example to stimulate mixing of fluids, or to reduce drag. In other cases, vortices have to be prevented, for example around aircraft, where they can reduce lift.

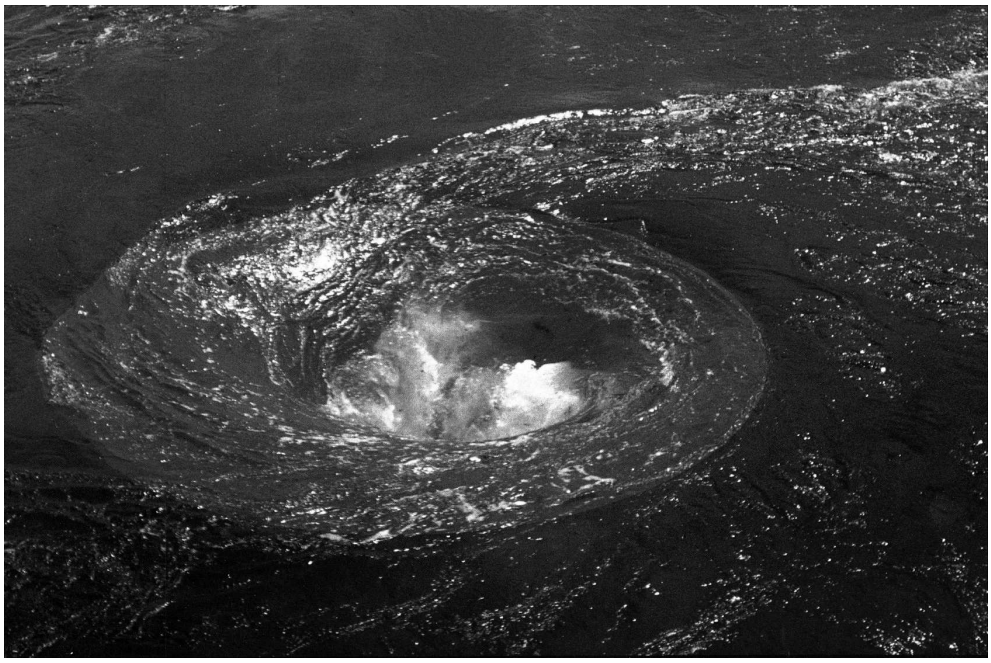


Figure 2.3: A vortex in water. (Source: WL | Delft Hydraulics)

There are many different definitions of vortices and likewise many different vortex detection algorithms. A distinction can be made in algorithms for finding vortex regions and algorithms that only find the vortex cores.

Other overviews of algorithms are given by Roth and Peikert [67] and by Banks and Singer [3].

There are a number of algorithms for finding *regions with vortices*:

- One idea is to find regions with a high vorticity magnitude. Vorticity is the curl of the velocity, that is, $\nabla \times \mathbf{v}$, and represents the local flow rotation, both in speed and direction. However, although a vortex may have a high vorticity

magnitude, the converse is not always true [109]. Villasenor and Vincent present an algorithm for constructing vortex tubes using this idea [95]. They compute the average length of all vorticity vectors contained in small-radius cylinders, and use the cylinder with the maximum average for constructing the vortex tubes.

- Another idea is to make use of helicity instead of vorticity [39, 108]. The helicity of a flow is the projection of the vorticity onto the velocity, that is $(\nabla \times \mathbf{v}) \cdot \mathbf{v}$. This way, the component of the vorticity perpendicular to the velocity is eliminated.

- Because swirling flow often swirls around areas of low pressure, this is another criterion that can be used to locate vortex cores [64].

- Jeong and Hussain define a vortex as a region where two eigenvalues of the symmetric matrix $S^2 + \Omega^2$ are negative, where S and Ω are the symmetric and antisymmetric parts of the Jacobian of the vector field, respectively [25]: $S = \frac{1}{2}(V + V^T)$, and $\Omega = \frac{1}{2}(V - V^T)$. This method is known as the λ_2 method.

The above methods may all work in certain simple flow data sets, but they do not hold, for example, in turbomachinery flows, which can contain strongly curved vortices [67].

There are also some algorithms specifically for finding *vortex core lines*:

- Banks and Singer use streamlines of the vorticity field, with a correction to the pressure minimum in the plane perpendicular to the vortex core [3].

- Roth and Peikert suggest that a vortex core line can be found where vorticity is parallel to velocity [67]. This sometimes results in coherent structures, but in most data sets it does not give the expected features.

- In the same article, Roth and Peikert suggest that, in linear fields, the vortex core line is located where the Jacobian has one real-valued eigenvector, and this eigenvector is parallel to the flow [67]. However, in their own application of turbomachinery flows, the assumption of a linear flow is too simple. The same algorithm is presented by Sujudi and Haimes [88].

- Recently, Jiang et al. presented a new algorithm for vortex core region detection [26], which is based on ideas derived from combinatorial topology. The algorithm determines for each cell if it belongs to the vortex core, by examining its neighbouring vectors.

A few of these algorithms will be reviewed in more detail.

Sujudi and Haimes developed an algorithm for finding the centre of swirling flow in 3D vector fields and implemented this algorithm in pV3 [88]. Although pV3 can

use many types of grids, the algorithm has been implemented for tetrahedral cells. When using data sets with other types of cells, these first have to be decomposed into tetrahedral cells. This is done for efficiency, because linear interpolation for the velocity can be used in the case of tetrahedral cells. The algorithm is based on critical-point theory and uses the eigenvalues and eigenvectors of the velocity gradient tensor or rate-of-deformation tensor. The algorithm works on each point in the data set separately, making it very suitable for parallel processing. The algorithm searches for points where the velocity gradient tensor has one real and two complex-conjugate eigenvalues and the velocity is in the direction of the eigenvector, corresponding to the real eigenvalue. The algorithm results in large coherent structures when a strong swirling flow is present, and the grid cells are not too large. The algorithm is sensitive to the strength of the swirling flow, resulting in incoherent structures or even no structures at all in weak swirling flows. Also, if the grid cells are large, or irregularly sized, the algorithm has difficulties finding coherent structures or any structures at all.

Kenwright and Haimes also studied the eigenvector method and concluded that it has proven to be effective in many applications [31]. The drawbacks of the algorithm are that it does not produce contiguous lines. Line segments are drawn for each tetrahedral element, but they are not necessarily continuous across element boundaries. Furthermore, when the elements are not tetrahedra, they have to be decomposed into tetrahedra first, introducing a piecewise linear approximation for a nonlinear function. Another problem is that flow features are found that are not vortices. Instead, swirling flow is detected, of which vortices are an example. However, swirling flow also occurs in the formation of boundary layers. Finally, the eigenvector method is sensitive to other nonlocal vector features. For example, if two axes of swirl exist, the algorithm will indicate a rotation that is a combination of the two swirl directions. The eigenvector method has successfully been integrated into a finite element solver for guiding mesh refinement around the vortex core [11].

Roth and Peikert have developed a method for finding core lines using higher-order derivatives, making it possible to find strongly curved or bent vortices [68]. They observe that the eigenvector method is equivalent to finding points where the acceleration $\mathbf{a}$ is parallel to the velocity $\mathbf{v}$, or equivalently, to finding points of zero curvature. The acceleration $\mathbf{a}$ is defined as:

$$\mathbf{a} = \frac{D\mathbf{v}}{Dt}, \tag{2.3}$$

where the notation $\frac{Df}{Dt}$ is used for the *derivative following a particle*, which is defined, in a steady flow, as $\nabla f \cdot \mathbf{v}$. Therefore:

$$\mathbf{a} = \frac{D\mathbf{v}}{Dt} = \nabla \mathbf{v} \cdot \mathbf{v} = J \cdot \mathbf{v}, \tag{2.4}$$

with J the Jacobian of $\mathbf{v}$, that is the matrix of its first derivatives.

Roth and Peikert improve the algorithm by defining vortex cores as points where

$$\mathbf{b} = \frac{D\mathbf{a}}{Dt} = \frac{D^2\mathbf{v}}{Dt^2} \tag{2.5}$$

is parallel to $\mathbf{v}$, that is, points of zero torsion. The method involves computing a higher-order derivative, introducing problems with accuracy, but it performs very well. In comparison with the eigenvector method, this algorithm finds strongly curved vortices much more accurately. Roth and Peikert also introduce two attributes for the core lines: the strength of rotation and the quality of the solution. This makes it possible for the user to impose a threshold on the vortices, to eliminate weak or short vortices. Peikert and Roth have also introduced a new operator, the "parallel vectors" operator [53], with which they are able to mathematically describe a number of previously developed methods under one common denominator. Using this operator they can describe methods based on zero curvature, ridge and valley lines, extremum lines and more.

Jiang et al. recently presented a new approach for detecting vortex core regions [26]. The algorithm is based on an idea which has been derived from Sperner's lemma in combinatorial topology, which states that it is possible to deduce the properties of a triangulation, based on the information given at the boundary vertices. The algorithm uses this fact to classify points as belonging to a vortex core, based on the vector orientation at the neighbouring points. In 2D, the algorithm is very simple and straightforward, and has only linear complexity. In 3D, the algorithm is somewhat more difficult, because it first involves computing the vortex core direction, and next, the 2D algorithm is applied to the velocity vectors projected onto the plane perpendicular to the vortex core direction. Still, also the 3D algorithm has only linear complexity.

The above described methods all use a local criterion for determining on a point-to-point basis where the vortices are located. The next algorithms use global, geometric criteria for determining the location of the vortices. This is a consequence of using another vortex definition.

Sadarjoen and Post present two geometric methods for extracting vortices in 2D fields [71]. The first is the curvature centre method. For each sample point, the algorithm computes the curvature centre. In the case of vortices, this would result in a high density of centre points near the centre of the vortex. The method works but has the same limitations as traditional point-based methods, with some false and some missing centres. The second method is the winding-angle method, which has been inspired by the work of Portela [54]. The method detects vortices by selecting and clustering looping streamlines. The winding angle α_w of a streamline is defined as the sum of the angles between the consecutive streamline segments. Streamlines are selected that have made at least one complete rotation, that is, $\alpha_w \geq 2\pi$. A second criterion checks that the distance between the starting and ending points is relatively

small. The selected streamlines are used for vortex attribute calculation. The geometric mean is computed of all points of all streamlines belonging to the same vortex. An ellipse fitting is computed for each vortex, resulting in an approximate size and orientation for each vortex. Furthermore, the angular velocity and rotational direction can be computed. All these attributes can be used for visualising the vortices. (See Figure 2.4.)

Figure 2.4: Flow in the Atlantic Ocean, with streamlines and ellipses indicating vortices. Blue and red ellipses indicate vortices rotating clockwise and counterclockwise, respectively [72]. See also colour Figure C.3.

2.4.2 Shock wave extraction

Shock waves are also important features in flow data sets, and can occur, for example, in flows around aircraft. (See Figure 2.5 on the next page.) Shock waves can increase drag and cause structural failure, and therefore, are important phenomena for study. Shock waves are characterised by discontinuities in physical flow quantities such as pressure, density and velocity. Therefore, shock detection is comparable to edge detection, and similar principles could be used as in image processing. However, in

numerical simulations, the discontinuities are often smeared over several grid points, due to the limited resolution of the grid.

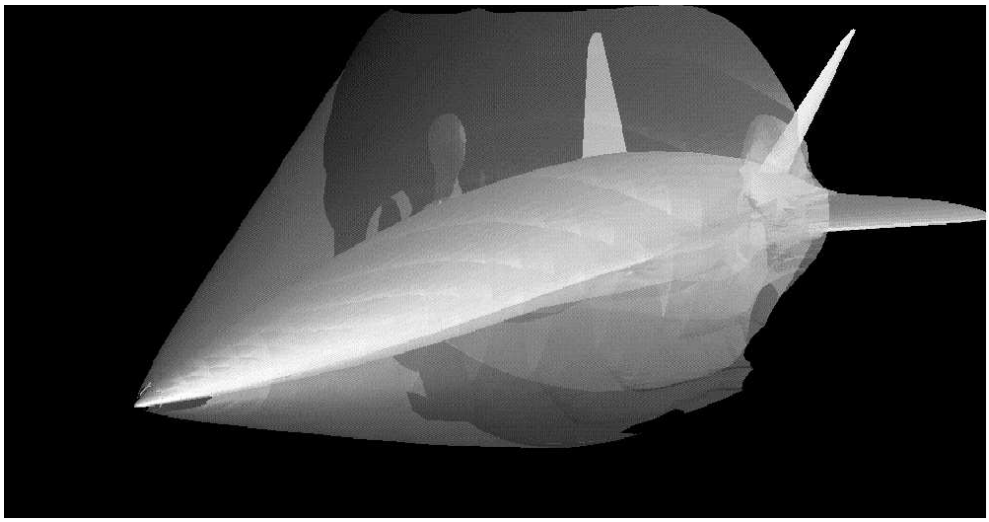


Figure 2.5: A shock wave around an aircraft. (Source: H.-G. Pagendarm)

Ma et al. have investigated a number of techniques for detecting and for visualising shock waves [44]. Detecting shocks in two dimensions has been extensively investigated [35, 45, 65]. However, these techniques are in general not applicable to shocks in three dimensions. They also describe a number of approaches for visualising shock waves. The approach of Haimes and Darmofal [19] is to create isosurfaces of the Mach number normal to the shock, using a combined density gradient/Mach number computation. Van Rosendale presents a two-dimensional shock-fitting algorithm for unstructured grids [65]. The idea relies on the comparison of density gradients between grid nodes.

Ma et al. compare a number of algorithms for shock extraction and also present their own technique [44]:

- The first idea is to create an isosurface of the points where the Mach number is one. However, this results in the sonic surface, which, in general, does not represent a shock.

- Theoretically, a better idea is to create an isosurface of the points where the normal Mach number is equal to one. However, if the surface is unknown, it is impossible to compute the Mach number, normal to the surface.

- This problem can be resolved, by approximating the shock normal with the density gradient, since a shock is also associated with a large gradient of the density. Therefore, $\nabla\rho$ is (roughly) normal to the shock surface. Thus, the algorithm computes the Mach number in the direction of, or projected onto, the density gradient. The shock surface is constructed from the points where this Mach number equals one. This algorithm is also used by Lovely and Haimes [43], but they define the shock region as the region within the isosurface of Mach number one, and use filtering techniques to reconstruct a sharp surface.

- Pagendarm presented an algorithm that searches for maxima in the density gradient [48]. The first and second derivatives of the density in the direction of the velocity are computed. Next, zero-level isosurfaces are constructed of the second derivative, to find the extrema in the density gradient. Finally, the first derivative is used to select only the maxima, which correspond to shock waves, and discard the minima, which represent expansion waves. This can be done by selecting only positive values of the first derivative. However, the second derivative can also be zero in smooth regions with few disturbances. In these regions the first derivative will be small, therefore, these regions can be excluded by discarding all points where the first derivative is below a certain threshold ϵ. Of course, this poses the problem of finding the correct ϵ. When the value is too small, erroneous shocks will be found, but if the value is too large, parts of the shocks could disappear. This algorithm can also be used for finding discontinuities in other types of scalar fields, and thus for finding other types of features.

- Ma et al. present an adapted version of this algorithm, which uses the normal Mach number to do the selection in the third step [44]. Again, in the first and second step, the zero-level isosurfaces of the second directional derivative of the density are constructed. But for discriminating shock waves from expansion waves and smooth regions, the normal Mach number is used. More precisely, those points are selected where the normal Mach number is close to one. Here also, a suitable neighbourhood of one has to be chosen.

2.4.3 Separation and attachment line extraction

Other features in flow data sets are separation and attachment lines on the boundaries of bodies in the flow. These are the lines where the flow abruptly moves away from or returns to the surface of the body. (See Figure 2.6 on the facing page.) These are important features in aerodynamic design because they can cause increased drag and reduced lift [66], and therefore, their occurrence should be prevented or at least minimised.

Helman and Hesselink use vector field topology to visualise flow fields [21]. In addition

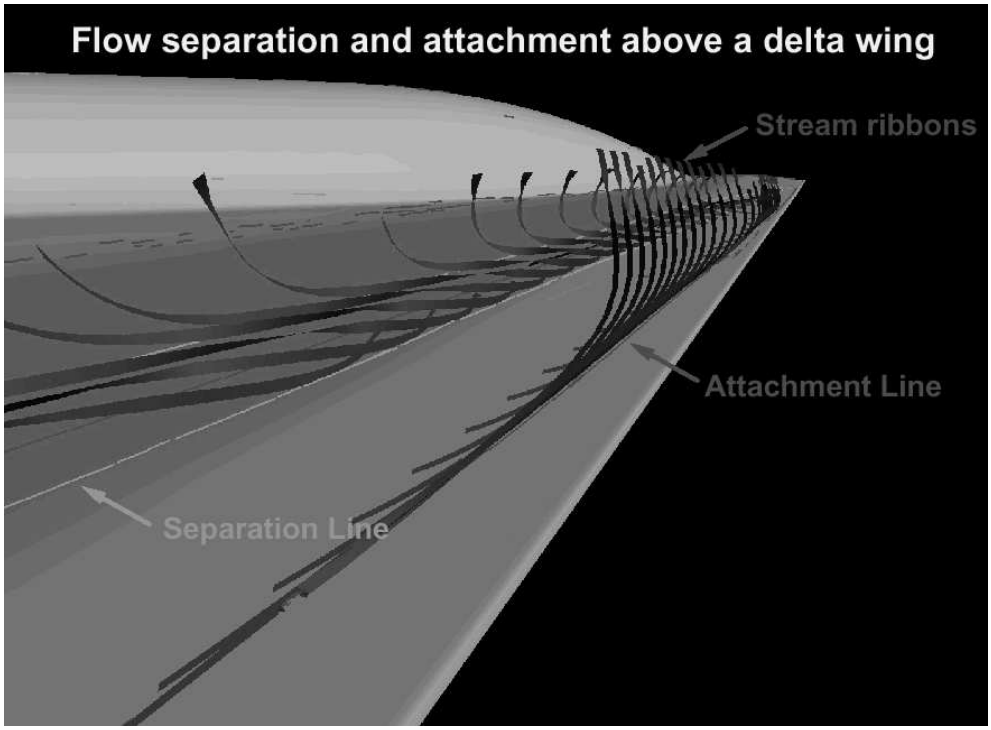


Figure 2.6: Separation and attachment lines on a delta wing. See also colour Figure C.4 (Source: D. Kenwright).

to the critical points, the attachment and detachment nodes on the surfaces of bodies determine the topology of the flow. (See Figure 2.7 on the next page.) The attachment and detachment nodes are not characterised by a zero velocity, because they only occur in flows with a no-slip condition, that is, all points on the boundaries of objects are constrained to have zero velocity. Instead, they are characterised by a zero tangential velocity. Therefore, streamlines impinging on the surface terminate at the attachment or detachment node, instead of being deflected along the surface.

Globus et al. designed and implemented a system for analysing and visualising the topology of a flow field with icons for the critical points and integral curves starting close to the critical points [14]. The system is also able to visualise attachment and detachment surfaces and vortex cores.

Pagendarm and Walter [49] and De Leeuw et al. [37] used skin-friction lines for visualising attachment and detachment lines in the blunt fin data set. For visualising these lines, the wall shear τ_w is computed, which is the flow velocity gradient per-

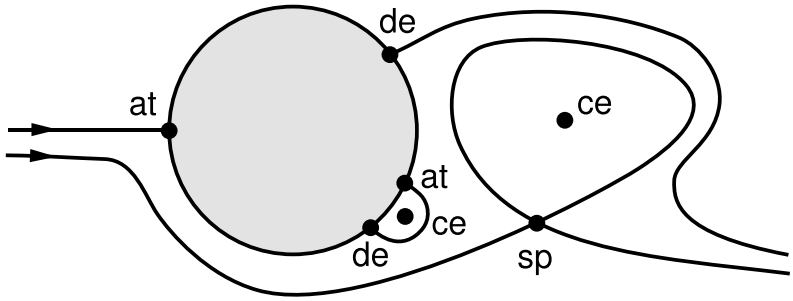


Figure 2.7: Vector field topology: a topological skeleton of a flow around a cylinder [21].

pendicular to the wall. Next, a standard streamline algorithm is used to integrate the skin-friction lines from the shear vector field. These skin-friction lines show the location of separation and attachment of the flow at the wall. (See Figure 2.8 on the facing page.)

Kenwright gives an overview of existing techniques for visualising separation and attachment lines and presents a new automatic feature detection technique for locating these lines, based on concepts from 2D phase plane analysis [30]. Some common approaches are:

- Particle seeding and computation of integral curves, such as streamlines and streaklines, which are constrained to the surface of the body. These curves merge along separation lines.
- Skin-friction lines can be used, analogous to surface oil flow techniques from wind tunnel experiments [49].
- Texture synthesis techniques can be used to create continuous flow patterns rather than discrete lines [37].
- Helman and Hesselink can generate separation and attachment lines from their vector field topology [21]. These lines are generated by integrating curves from the saddle and node type critical points on the surface in the direction of the real eigenvector. However, only closed separations are found, that is, curves that start and end at critical points.

Open separation does not require separation lines to start or end at critical points, and is therefore not detected using flow topology. Open separation has been observed in experiments, but had not previously been studied in flow simulations. However, the algorithm presented by Kenwright does detect both closed and open separation

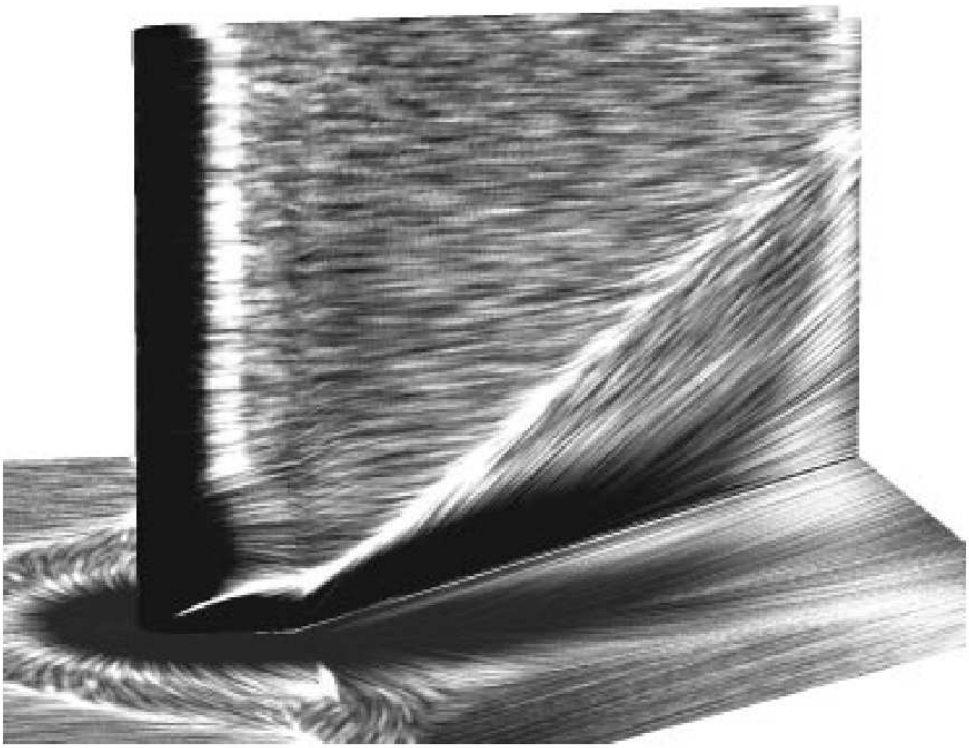


Figure 2.8: Skin-friction on a blunt fin from a flow simulation at Mach 5, visualised with spot noise [37].

lines. The theory for this algorithm is based on concepts from linear phase plane analysis. It is assumed that the computational domain on the surface can be subdivided into triangles and the vector components are given at the vertices. The algorithm is executed for each triangle, making it suitable for parallelisation. For each triangle, a linear vector field is constructed satisfying the vectors at the vertices. If the determinant of the Jacobian matrix is nonzero, the algorithm continues by calculating the eigenvalues and eigenvectors of the Jacobian. Every triangle has a critical point somewhere in its vector field. The linear vector field is translated to this critical point and the coordinate system is changed so that the eigenvectors are orthogonal. This (x, y) plane is also referred to as the Poincaré phase plane. (See Figure 2.9 on the next page.) By computing tangent curves in the phase plane, we obtain the phase portrait of the system. For a saddle, the tangent curves or streamlines converge along the x and y axes. For a repelling node, they converge along the y axis and for an attracting node, they converge along the x axis. If the phase portrait is a saddle or a repelling

node, the intersection of the y axis with the triangle is computed. If it intersects, the line segment will form part of an attachment line. If the phase portrait is a saddle or an attracting node, the intersection of the x axis with the triangle is computed, and if it does intersect, the line segment will form part of a separation line.

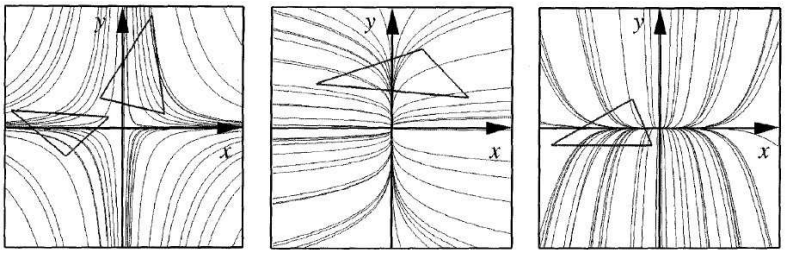


Figure 2.9: Three phase portraits, for a saddle, repelling node and attracting node. The intersections of the triangles with the axes contribute line segments to attachment or separation lines [30].

A problem with this algorithm is that disjointed line segments are computed instead of continuous attachment and separation lines. Other problems occur when the flow separation or attachment is relatively weak, or when the assumption of locally linear flow is not correct.

Kenwright et al. present two algorithms for detecting separation and attachment lines [32]. The first is the algorithm discussed above, the second is the parallel vector algorithm. Both algorithms use eigenvector analysis of the velocity gradient tensor. However, the first is element-based and results in disjointed line segments, while the second is point-based and will result in continuous lines.

In the parallel vector algorithm, points are located where one of the eigenvectors $\mathbf{e}_i$ of the gradient $\nabla\mathbf{v}$ is parallel to the vector field $\mathbf{v}$, that is, points where the streamline curvature is zero, or in formula:

$$\mathbf{e}_i \times \mathbf{v} = \mathbf{0}. \tag{2.6}$$

The velocity vectors and the eigenvectors can be determined at the vertices of the grid and interpolated within the elements. At the vertices, $\mathbf{e}_i \times \mathbf{v}$ is calculated for both eigenvectors, but only if both eigenvectors are real, that is, the classification of $\nabla\mathbf{v}$ at the vertex is either a saddle or a node. If the cross product $\mathbf{e}_i \times \mathbf{v}$ changes sign across an edge, that means an attachment or separation line intersects the edge. The intersection point can then be found by interpolation along the edge. The attachment and separation lines can be constructed by connecting the intersection points in each element. The distinction between attachment and separation can be made easily, because attachment will occur where $\mathbf{v}$ is parallel to the smallest $\mathbf{e}_i$ and separation where $\mathbf{v}$ is parallel to the largest $\mathbf{e}_i$. Another set of lines is detected with this

algorithm, the inflection lines, where one of the eigenvectors is locally parallel to the velocity vector, but the line itself is not an asymptote of neighbouring streamlines. (See Figure 2.10.) These inflection lines can easily be filtered out by checking if:

$$\nabla(\mathbf{e}_i \times \mathbf{v}) \cdot \mathbf{v} = \mathbf{0}. \tag{2.7}$$

This will not be true for inflection lines.

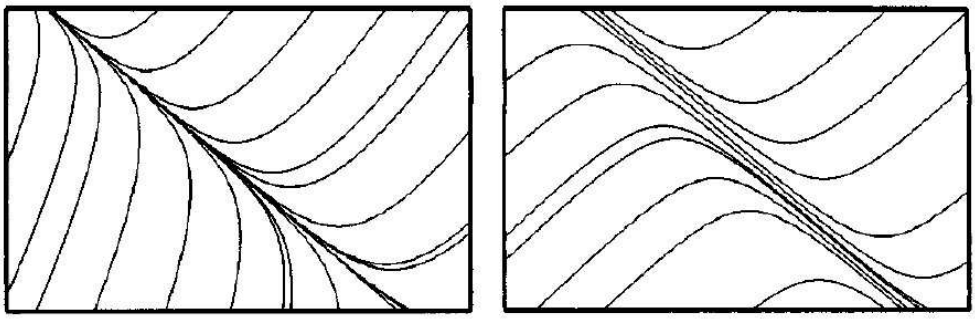


Figure 2.10: The vector field in the left image contains a separation line; the field in the right image contains an inflection line [32].

Both algorithms discussed by Kenwright et al. correctly identify many separation and attachment lines, but may fail in identifying curved separation lines [32]. The parallel vector algorithm will result in continuous lines, whereas the phase plane algorithm results in discontinuous line segments. Both algorithms do detect open separation lines, which do not start or end at critical points.

2.5 Feature tracking and event detection

In time-dependent data sets, features are objects that evolve in time. Determining the correspondence between features in successive time steps, that actually represent the same object at different times, is called the *correspondence problem*. Feature tracking is involved with solving this correspondence problem. The goal of feature tracking is to be able to describe the evolution of features through time. During the evolution, certain *events* can occur, such as the interaction of two or more features, or significant shape changes of features. Event detection is the process of detecting such events, in order to describe the evolution of the features even more accurately.

There are a number of approaches to solving the correspondence problem. Features can be extracted directly from the spatio-temporal domain, thereby implicitly solving the correspondence problem. Or, when feature extraction is done in separate time

steps, the correspondence can be solved based on region correspondence, or based on attribute correspondence.

2.5.1 Feature extraction from the spatio-temporal domain

It is possible to perform feature extraction in 3D or 4D space-time. Tricoche et al. present an algorithm for tracking of two-dimensional vector field topologies by interpolation in 3D space-time [91]. Bajaj et al. present a general technique for hypervolume visualisation [1]. They describe an algorithm to visualise arbitrary n-dimensional scalar fields, possibly with one or more time dimensions. Weigle and Banks extract features by isosurfacing in four-dimensional space-time [101]. This is conceptually similar to finding overlapping features in successive time steps. See also the next Section (2.5.2), about region correspondence. Bauer and Peikert perform tracking of features in (4D or 5D) scale-space [4]. The idea is that the original data is smoothed using a Gaussian kernel. The standard deviation σ of this kernel can be any positive number, and is represented on the scale axis. Together with the normal 3D spatial axes, and possibly one time axis, this scale axis spans the scale-space. In the article, the focus is on line-type features, and specifically vortex cores, but that is just their main application, and not inherent to the algorithm. In 5D scale-space, it is possible to track features not only along the time axis, but also along the scale axis.

2.5.2 Region correspondence

Region correspondence involves comparing the regions of interest obtained by feature extraction. Basically, the binary images from successive time steps, containing the features found in these time steps, are compared on a cell-to-cell basis. Correspondence can be found using a minimum distance or a maximum cross-correlation criterion [16] or by minimising an affine transformation matrix [29]. It is also possible to extract isosurfaces from the four-dimensional time-dependent data set [101], where time is the fourth dimension. The correspondence is then implicitly determined by spatial overlap between successive time steps. This criterion is simple, but not always correct, as objects can overlap but not correspond, or correspond but not overlap. Silver and Wang explicitly use the criterion of spatial overlap instead of creating isosurfaces in four dimensions [83, 84]. They prevent correspondence by accidental overlap, by checking the volume of the corresponding features and taking the best match. This is also the idea of attribute correspondence, which is discussed next. By using spatial overlap, certain events are implicitly detected, such as a *bifurcation* when a feature in one time step overlaps with two features in the next time step. Event detection is also discussed more extensively later, in Section 2.5.4.

2.5.3 Attribute correspondence

With attribute correspondence, the comparison of features from successive frames is performed on the basis of the attributes of the features, such as the position, size, volume, and orientation. These attributes can be computed in the feature extraction phase, (see Section 2.3.4,) and can be used for description and for visualisation of the features, and also for feature tracking, as described here. The original grid data is not needed anymore. Samtaney et al. use the attribute values together with user-provided tolerances to create correspondence criteria [75]. For example, for position the following criterion could be used:

$$dist(pos(O_{i+1}), pos(O_i)) \leq T_{dist}, \tag{2.8}$$

where $pos(O_i)$ and $pos(O_{i+1})$ are the positions of the objects in time steps i and $i+1$, respectively, and T_{dist} is the user-provided tolerance. For scalar attributes, the difference or the relative difference could be used. For example, to test the relative difference of the volume, the following formula can be used:

$$\frac{vol(O_{i+1}) - vol(O_i)}{\max(vol(O_{i+1}), vol(O_i))} \leq T_{vol}, \tag{2.9}$$

where $vol(O_i)$ and $vol(O_{i+1})$ are the volumes of the features in the two time steps, and T_{vol} is the tolerance given by the user. Events such as a bifurcation can also be tested. If a feature in time step i splits into two features in time step $i+1$, the total volume after the event has to be approximately the same as before the event. The same formula can be used as for the normal volume test, except that $vol(O_{i+1})$ in this case equals the sum of the volumes of the separate features. The position criterion in case of a bifurcation event could involve the weighted average of the individual positions after the event, where the positions are weighed with the volume:

$$dist(pos(O_i), \frac{\sum(vol(O_{i+1}) \cdot pos(O_{i+1}))}{\sum(vol(O_{i+1}))}) \leq T_{dist}, \tag{2.10}$$

where O_{i+1} now represents all objects in time step $i+1$ that are involved in the event.

Reinders et al. describe an algorithm for feature tracking, that is based on prediction and verification [59, 60]. This algorithm is based on the assumption that features evolve predictably. That means, if a part of the evolution of a feature (*path*) has been found, a prediction can be made into the next time step (*frame*). Then, in that next time step, a feature is sought, that corresponds to the prediction. (See Figure 2.11 on the next page.) If a feature is found that matches the prediction within certain user-provided tolerances, the feature is added to the evolution and the search is continued to the next time step. When no more features can be added to the path, a new path is started. In this manner, all frames are searched for starting points, both in forward and backward time direction, until no more paths can be created. A path is

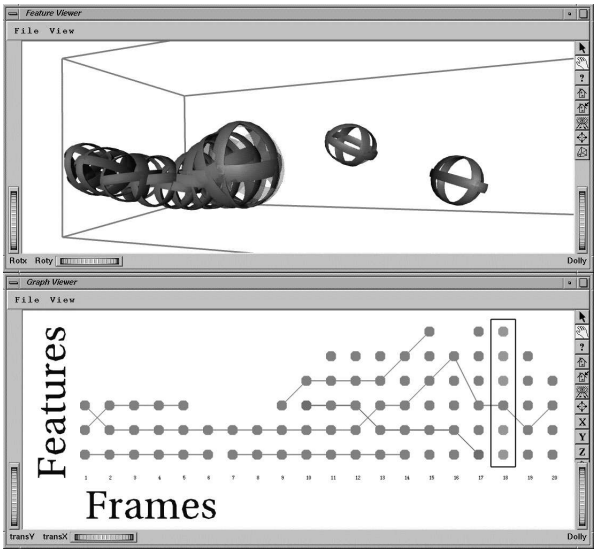


Figure 2.11: One step during feature tracking. A path is shown with its prediction, and three candidates in the next time step [60]. See also colour Figure C.5.

started by trying all possible combinations of features from two consecutive frames and computing the prediction to the next frame. Then, the prediction is compared to the candidate features in that frame. If there is a match between the prediction and the candidate, a path is started. To avoid any erroneous or coincidental paths, there is a parameter for the minimal path length, which is usually set to 4 or 5 frames. A candidate feature can be defined in two ways. All features in the frame can be used as candidates, or only unmatched features can be used, that is, those features that have not yet been assigned to any path. The first definition ensures that all possible combinations are tested and that the best correspondence is chosen. However, it could also result in features being added to more than one path. This has to be resolved afterwards. Using the second definition is much more efficient, because the more paths are found, the fewer unmatched features require testing. However, in this case, the results depend on the order in which the features are tested. This problem can be solved by starting the tracking process with strict tolerances and relaxing the tolerances in subsequent passes.

The prediction of a feature is constructed by linear extrapolation of the attributes of the features from the last two frames. Other prediction schemes could also be used, for example, if a priori knowledge of the flow is available.

The prediction is matched against real features using correspondence criteria, similar to the ones used by Samtaney et al. as discussed above [75]. For each attribute of the

features, a correspondence function can be created, which returns a positive value for a correspondence within the given tolerance, with a value of 1 for an exact match, and a negative value for no correspondence. Each correspondence function is assigned a weight, besides the tolerance. Using this weight, a weighted average is calculated of all correspondence functions, resulting in the correspondence factor between the two features. For this correspondence factor, the same applies as for the separate correspondence functions, that is, a positive value indicates a correspondence, with 1 indicating a perfect match. A negative correspondence factor means no match.

2.5.4 Event detection

After feature tracking has been performed, event detection is the next step. Events are the temporal counterparts of spatial features in the evolution of features. For example, if the path or evolution of a feature ends, it can be interesting to determine why that happens. It could be that the feature shrinks and vanishes, or that the feature moves to the boundary of the data set and disappears, or that the feature merges with another feature and the two continue as one. Samtaney et al. introduced the following events: continuation, creation, dissipation, bifurcation, amalgamation [75]. (See Figure 2.12.) Reinders et al. developed a feature tracking system that is able to

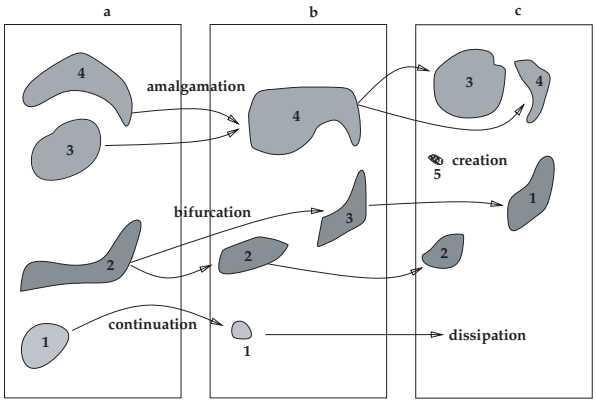


Figure 2.12: The different types of events as introduced by Samtaney et al. [75].

detect these and other events [60]. The terminology they use is *birth* and *death* instead of creation and dissipation, and *split* and *merge* for bifurcation and amalgamation. Furthermore, they can detect *entry* and *exit* events, where a feature moves beyond the boundary of the data set. Finally, for a specific, graph-type feature, the system is able to detect changes in topology. It discriminates *loop* and *junction* events. (See Figure 2.13 on the next page.) Many other types of events can be envisioned, but for

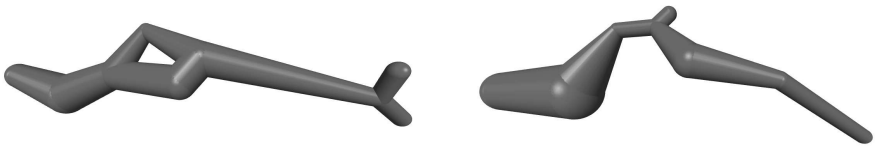


Figure 2.13: A loop event has occurred. In the left image, the feature contains a loop, in the right image, the next frame, the loop has disappeared [57].

each type specific detection criteria have to be provided.

For event detection, just as for feature tracking, only the feature attributes are used. Analogous to the correspondence functions, for event detection, event functions are computed. For example, to detect a death event, two conditions must hold. First, the volume of the feature must decrease. And second, the volume of the prediction must be very small or negative. The event function for this event returns a positive value if the volume of the prediction is within the user-provided tolerance, and is equal to one if the volume of the prediction is negative. If the volume is not within the tolerance, the returned value will be negative. The event functions for the separate attributes are combined into a single factor, which determines if the event is a death event. A birth event can be detected by doing the same tests in the backward time direction.

Similarly, the tests for split and merge events, and for entry and exit events are each other's reverse in time.

2.6 Visualisation of features and events

The final step in the feature extraction pipeline is, of course, the visualisation of the features. A number of techniques will be covered in this Section. The most straightforward visualisation is to show the nodes in the data set, that have been selected in the first step of the feature extraction pipeline. This step results in a binary data set, with each value indicating whether the corresponding node has been selected or not. This binary data set can be visualised, for example, with crosses at the selected nodes. In Figure 2.14 on the facing page, such a visualisation is shown. The visualisation is of a simulation of the flow behind a backward-facing step. The feature that is visualised here is a recirculation zone, behind the step. The points were selected with the criterion: normalised helicity $H > 0.6$.

Another simple visualisation technique is to use isosurfaces. This can be done on the binary data set, resulting from the selection step, or, if the selection expression is a simple threshold, directly on the original data set. This results in isosurfaces

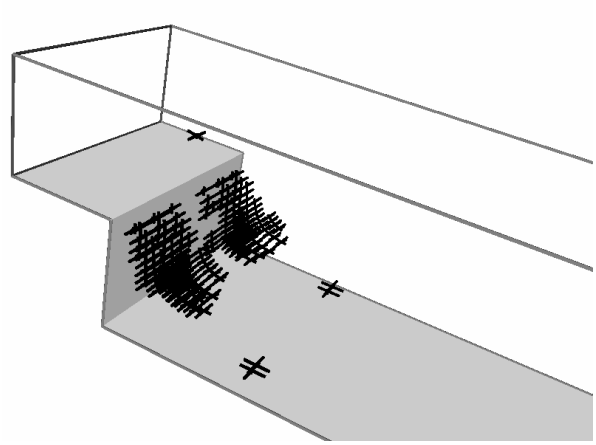


Figure 2.14: Visualisation of the selected points in the backward-facing-step data set [73].

enclosing the selected regions.

Also, other standard visualisation techniques can be used in combination with the Boolean data set resulting from the selection step. For example, in a 3D flow data set, using the standard methods for seeding streamlines or streamtubes, will not provide much information about the features and will possibly result in visual clutter. However, if the selected points are used to seed streamlines, both backward and forward in time, this can provide useful information about the features and their origination. See Figure 2.15 on the next page, for an example, where two streamtubes are shown in the backward-facing-step data set. The radius of the tubes is inversely proportional to the square root of the local velocity magnitude, and the colour of the tubes corresponds to the pressure.

If, instead of the separate selected points, the attributes are used, that have been computed in the feature extraction process, then parametric icons can be used for visualising the features.

If an ellipsoid fitting of the selected clusters has been computed, there are three attribute vectors: the centre position, the axis lengths, and the axis orientations, which can be mapped onto the parameters of an ellipsoid icon. This is a simple icon, but very efficient and accurate. It can be represented with 9 floating-point values, and is therefore space-efficient. Furthermore, it can be very quickly visualised, and although it is simple, it gives an accurate indication of the position and volume of a feature. In Figure 2.16 on page 39, an ellipsoid fitting is computed from the selected points in Figure 2.14. In Figure 2.17, vortices are shown from a CFD simulation

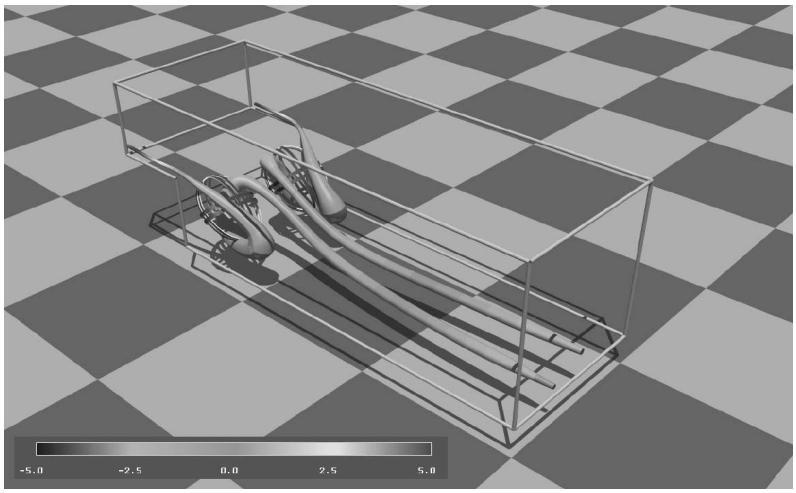


Figure 2.15: Visualisation with streamtubes of the recirculation in the backward-facing-step data set [99]. See also colour Figure C.6.

with turbulent vortex structures. The features have been selected by a threshold on vorticity magnitude. They are being visualised with isosurfaces and ellipsoids. It is clearly visible that, in this application, with the strongly curved features, the ellipsoids do not give a good indication of the shape of the features. But, as mentioned above, the position and volume attributes of the ellipsoids will be accurate, and can be used for feature tracking.

In Figure 2.18 on page 41, the flow past a tapered cylinder is shown. Streamlines indicate the flow direction, and rotating streamlines indicate vortices. The vortices are selected by locating these rotating streamlines, using the winding-angle method [71]. Ellipses are used to visualise the vortices, with the colour indicating the rotational direction. Green means clockwise rotation, red means counterclockwise rotation. The slice is coloured with λ_2, which is the second-largest eigenvalue of the tensor $S^2 + \Omega^2$. (See Section 2.4.1.) The tapered cylinder data set consists of a number of horizontal slices, such as the one in Figure 2.18. Figure 2.19 shows an image of the three-dimensional vortices, which have been constructed from the ellipses extracted in each slice [61].

For the 3D vortices in Figure 2.17, an other type of icon has to be used, if we want to visualise the strongly curved shape of the features. Reinders et al. present the use of skeleton graph descriptions for features, with which they can create icons that accurately describe the topology of the features, and approximately describe the shape of the features [58]. Compare the use of ellipsoid icons with the use of skeleton icons in Figure 2.20 on page 43.

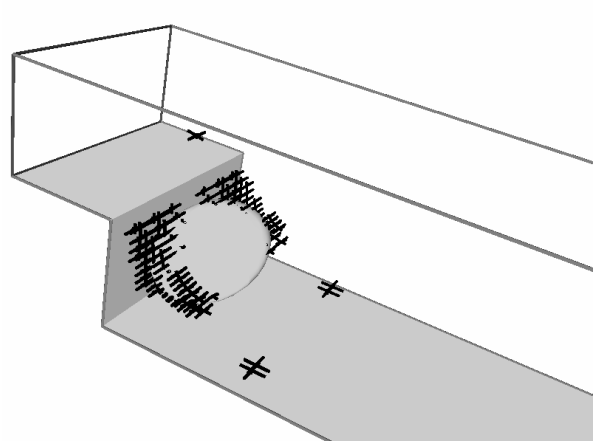


Figure 2.16: An ellipsoid fitting computed from the selected points in the backward-facing-step data set [73].

For visualising the results of feature tracking, it is of course essential to visualise the time dimension. The most obvious way is to animate the features, and to give the user the opportunity to browse through the time steps, both backward and forward in time. Figure 2.21 on page 43 shows the player from the feature tracking program, developed by Reinders [60]. On the left of the image, the graph viewer is shown, which gives an abstract overview of the entire data set, with the time steps on the horizontal axis, and the features represented by nodes, on the vertical axis. The correspondences between features from consecutive frames are represented by edges in the graph, and therefore, the evolution of a feature in time, is represented by a path in the graph. On the right of the image, the feature viewer is shown, in which the feature icons from the current frame are displayed. Also, a control panel is visible, with which the animation can be started, paused, and played forward and backward.

The graph viewer can also be used for visualising events [57]. For each event, a specific icon has been created, which is mapped onto the nodes of the graph, so that the user can quickly see which events occur where, and how often they occur. In Figure 2.22 on page 44, the graph viewer is shown, with a part of the graph, containing a number of events. Each event is clearly recognisable by its icon. In Figure 2.23, two frames are shown, between which a split event has occurred. In both frames, the features are shown with both ellipsoid and skeleton icons. The advantage of the use of skeleton icons in this application is obvious. Because the shape of the features is much more accurately represented by the skeleton icons, changes in shape and events such as these are much more easily detected.

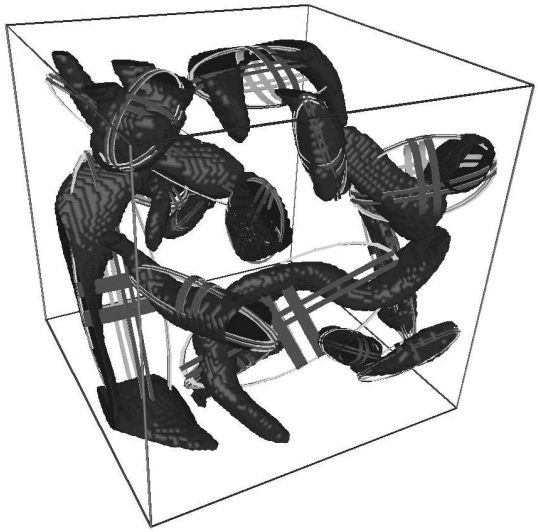


Figure 2.17: Vortices in a data set with turbulent vortex structures, visualised using isosurfaces and ellipsoids [57].

2.7 Conclusions and future prospects

Feature extraction is selection and simplification based on content: extracting relevant high-level information from a data set, visualising the data from a problem-oriented point of view. This leads to a large reduction of the data size, and to fully or semi-automatic generation of simple and clear images. The techniques are generally very specific for a certain type of problem (such as vortex detection), the relation with the original raw data is indirect, and the reduction is achieved at the cost of loss of other information, which is considered not relevant for the purpose. But the techniques generalise well to analysis of time-dependent data sets, leading to condensed episodic visual summaries.

A good possibility is combining feature extraction techniques with direct or geometric techniques. For example, selective visualisation has been used effectively with streamline generation (Figure 2.15), to place seed points in selected areas, and show important structures with only a small number of streamlines. Combining simple advection-based techniques with iconic feature visualisation can also clarify the relation between the raw data and the derived information used in feature detection (Figure 2.18). The work of visualisation and simulation experts will become inseparable in the future: the distinction between simulation and visualisation will be increasingly blurred. A good example is the tracking of phase fronts (separation be-

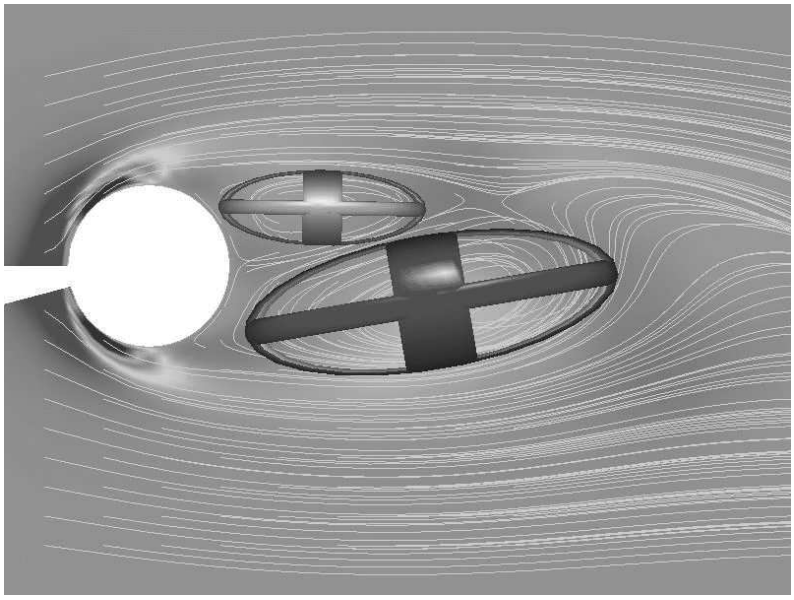


Figure 2.18: Vortices behind a tapered cylinder. The colour of the ellipsoids represents the rotational direction [71]. See also colour Figure C.7.

tween two different fluids in multi-fluid flows) using level set methods [77], where the feature extraction is a part of both simulation and visualisation.

How about practical application? Feature-based techniques have been incorporated in commercial visualisation systems [1]. The practical use of flow visualisation is most effective when visualisation experts closely cooperate with fluid dynamics experts. This is especially true in feature-based visualisation, where developing detection criteria is closely connected to the physical phenomena studied. But also other disciplines can contribute to this effort: mathematicians, artists and designers, experimental scientists, image processing specialists, and also perceptual and cognitive scientists [55].

In feature-based visualisation, the following areas need additional work:

- interactive techniques to support extraction and tracking of features [12];
- detection and tracking of new types of features, such as recirculation zones, boundary layers, phase fronts, and mixing zones, and detection of new types of events;
- comparative visualisation based on quantitative feature comparison;

[1] `http://www.ensight.com/products/flow-feature.html`

Figure 2.19: 3D Vortex structures behind a tapered cylinder [61]. The number and curvature of the spokes indicate the rotational speed and direction, respectively.

- topological analysis: extension to finding separation surfaces in 3D and to time-dependent flows;
- image processing: adaptation of image segmentation and filtering techniques to irregular grids and use with vector fields;
- online steering of large simulations based on feature extraction and event detection.

Overlooking the whole landscape of flow visualisation techniques, we can say that visualisation of 2D flows has reached a high level of perfection, and for visualisation of 3D flows a rich set of techniques is available. In the future, we will concentrate on techniques that scale well with ever increasing data set sizes, and therefore simplification, selection, and abstraction techniques will get more attention.

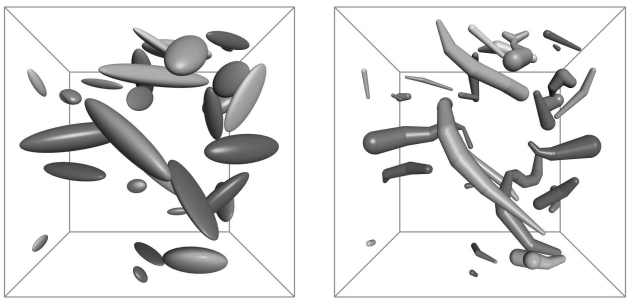


Figure 2.20: Turbulent vortex structures represented by ellipsoid icons (left) and skeleton icons (right) [57].

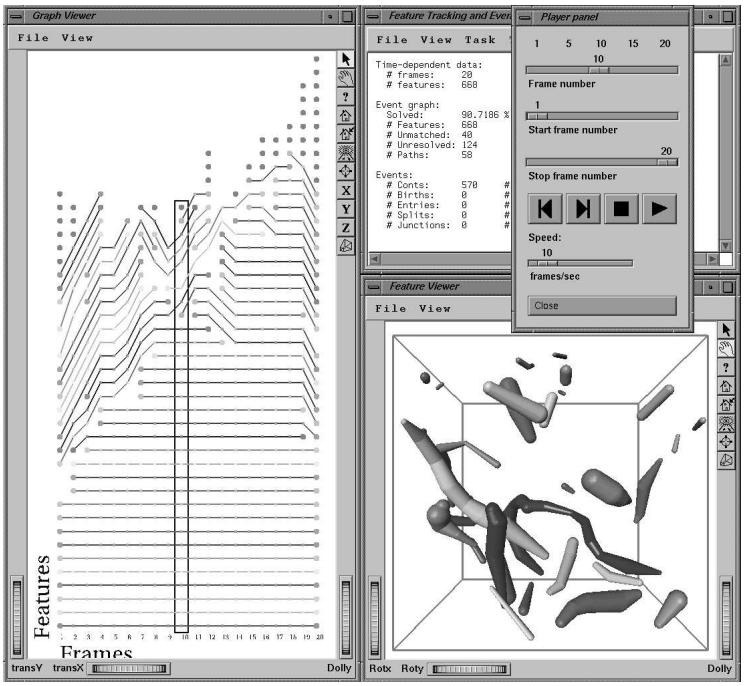


Figure 2.21: Playing through the turbulent vortex data set. See also colour Figure C.8.

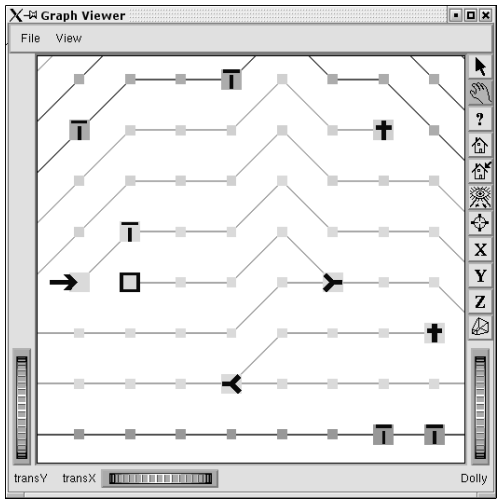


Figure 2.22: Events are visualised in the graph viewer with special, characteristic icons.

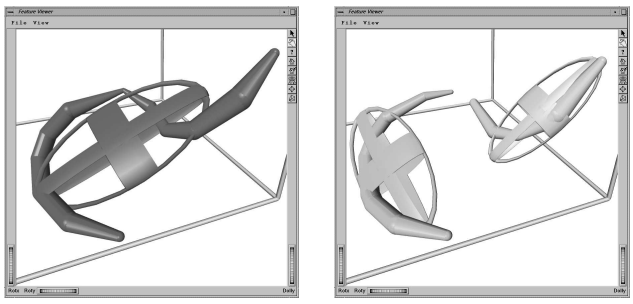


Figure 2.23: A split event, before (left) and after (right). The features are visualised with both an ellipsoid and a skeleton icon [57].

CHAPTER 3

Data structures for very large data handling

3.1 Introduction

As was described in Chapter 1, the process from multi-phase flow simulation to visualisation can be split into a number of steps. One of the steps is the detection and extraction of the surface of the phase front. In the previous Chapter several techniques were discussed from feature-based flow visualisation. Techniques similar to the ones for shock wave extraction, described in Section 2.4.2, would have to be developed for detecting the phase front. Because a phase front is characterised by a sudden jump in physical quantities such as density, it is comparable in nature to a shock wave. Therefore, similar techniques could be used for extraction and tracking of both types of features.

However, after about a year into this project, it became apparent that a separate feature detection step would no longer be required. Our co-researchers within the project developed a new mass-conserving Level-Set method, which includes an implicit surface representation of the phase front [94]. During the simulation, this surface representation will be maintained, and it will be stored to disk, together with the flow data from the simulation. No special feature detection techniques will be needed for this phase front, because it can easily be described as an isosurface.

Whereas the first method would result in a large data reduction, because of the

feature-based approach, this second method still results in huge data sets. The problem no longer lies in detection and extraction of the surface as well as in handling of the very large data sets.

Therefore, referring to the list of approaches in Section 1.2, we have decided to move away from the feature-based techniques and look more specifically into the other approaches.

In the current Chapter a number of techniques from the other approaches will be discussed in more detail. In Section 3.2 two examples will be given of *compression* techniques. The algorithms discussed in Section 3.3 can be classified as *multi-resolution data structures*. Finally, in Section 3.4 a *fast access data structure* will be discussed.

3.2 Compression techniques

As was already introduced in Chapter 1, compression can be described as *statistics-based* data reduction. Many data compression algorithms originate from the field of image processing.

Examples of compression techniques include run-length encoding (RLE), difference or delta encoding, entropy encoding (e.g. Huffman coding [22]), transform coding (e.g. using the discrete cosine transform as in JPEG [28] and MPEG [46, 47] or the wavelet transform as in JPEG-2000) and combinations of these [7].

As an example of a method that uses delta encoding, Ibarria et al. use the Lorenzo predictor [23] to estimate data values in a 4D data set from their spatial and temporal neighbours. This technique will be discussed in more detail in the following Subsection.

A second example, discussed in Subsection 3.2.2 uses wavelets to compress a data set up to a user-specified level. There is a huge amount of literature on wavelets and their applications. A short introduction to wavelets will be presented in Subsection 3.2.2. For a more elaborate introduction to the subject, see for example the book by Chui [9]. An early example of the use of wavelets for multi-resolution volume rendering was presented by Westermann [102].

3.2.1 Out-of-core compression using the Lorenzo predictor

Ibarria et al. presented a method for compressing and decompressing very large (n-dimensional) scalar data sets, requiring only a small amount of memory, using a simple prediction scheme [23].

They introduce the Lorenzo predictor, which estimates the scalar value at a certain

grid point from the values that have already been processed. More precisely, the scalar value at the corner of an n-dimensional cube of grid points will be predicted from the values of its $2^n - 1$ immediate neighbours — the other corners of the cube.

The predictor uses a very simple formula. The estimated value is a weighted sum of the values of the neighbours. The weights are alternatingly $+1$ and -1, depending on the degree of the neighbour. See Figure 3.1: immediate neighbours have weight $+1$, second degree neighbours have weight -1, et cetera.

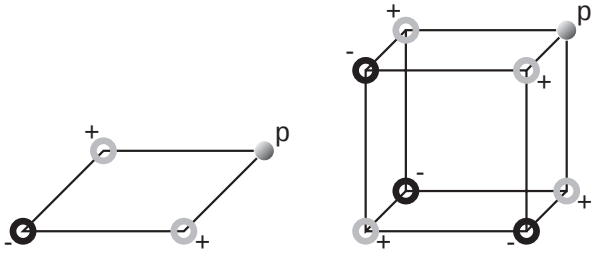


Figure 3.1: The Lorenzo predictor estimates the value at point p from a weighted sum of its neighbours. The weights are alternatingly positive and negative, depending on the degree of the neighbour [23].

The estimates are exact when the scalar field corresponds to an implicit polynomial of degree $n-1$. Otherwise, the residuals (the differences between the real and predicted values) will probably still be relatively small, and can therefore be encoded using a smaller number of bits than the original data.

Because the predictor only uses the grid points at the corner of an n-dimensional unit cube, a small amount of memory suffices to process the data. Both during compression and decompression, the data is visited in scan line order. The amount of memory that is needed for the predictor, (the so-called *footprint*,) is slightly more than one $(n-1)$-dimensional slice. See Figure 3.2 on the following page. Therefore, this method, although presented here as a compression algorithm, could also be used for out-of-core processing.

3.2.2 Wavelet-based multi-resolution modelling

Contrary to what the name suggests, this technique is not really a multi-resolution but rather a compression technique. Therefore, I have categorised it in this instead of the next Section. This technique, presented by Baldwin et al., uses wavelets to create a compressed representation of data sets [2].

Wavelets are mathematical functions that can be used for frequency analysis of signals

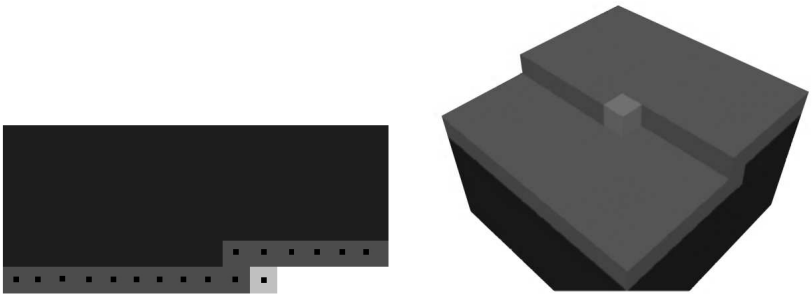


Figure 3.2: The footprint in 2D and 3D [23].

or data. More general than the Fourier transform, the wavelet transform uses functions that are localised in frequency *as well as* in space. Unlike the Fourier transform, with only the sine and cosine functions as basis functions, the wavelet transform has an infinite set of possible basis functions. For example, the Haar wavelet [17], which is the oldest and also the simplest possible wavelet, can be described as a step function $f(x)$ with

$$f(x) = \begin{cases} 1 & 0 \leq x < \frac{1}{2}, \\ -1 & \frac{1}{2} \leq x < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

See also Figure 3.3.

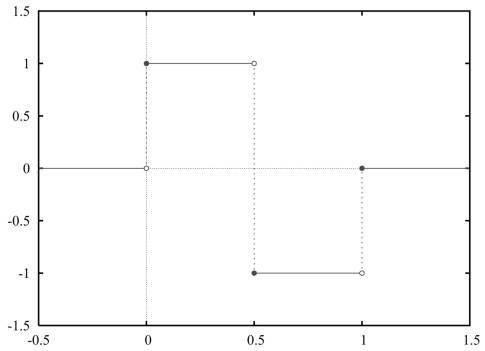


Figure 3.3: The Haar wavelet.

The signal that is to be analysed is processed at different scales or resolutions and with different windows. For each of these resolutions and windows, the basis function is scaled and shifted and convolved with the data. The result is that the data can be represented by a number of coefficients. In reconstruction, the more coefficients are

used, the more details (high frequencies) in the result. With only a small number of coefficients, still a good result can be achieved.

This technique by Baldwin et al. also results in a number of coefficients. These coefficients can be compressed up to a user-specified level:

- the coefficients can be chosen up to a user-specified amount, thereby assuring a certain compression ratio;
- the (sorted) coefficients can be chosen up to a user-specified relative error;
- the coefficients can be chosen that are greater than a certain user-specified minimum.

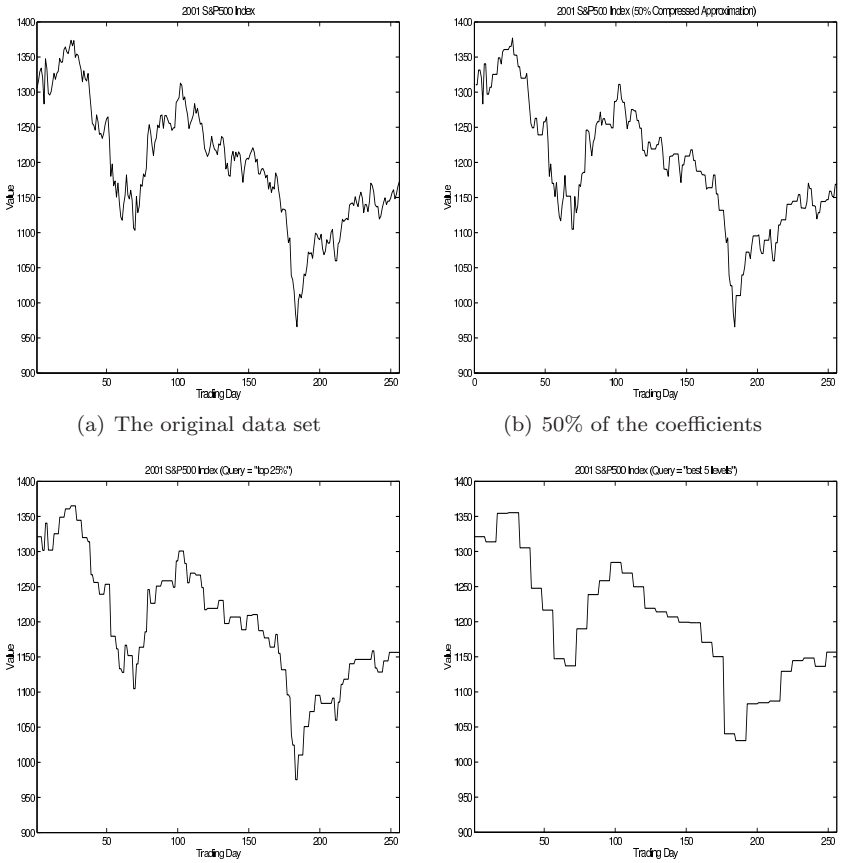
During decompression or reconstruction different degrees of approximation can be achieved by varying the number of wavelet coefficients used.

- First, the data can be reconstructed using all wavelet coefficients. This is in fact a decompression of the data.
- The data can be reconstructed with a user-specified percentage of the available (sorted) wavelet coefficients.
- The data can be reconstructed using the most significant coefficients up to a user-specified relative error.
- The data can be reconstructed using the most significant levels of the wavelet decomposition.
- Finally, the data can be reconstructed using only the coefficients that affect a certain spatial location.

Both the second and third method can be implemented in a progressive manner. Starting from the most important coefficient and thus the coarsest image, coefficients are added, progressively improving the accuracy and adding details.

Depending on the amount of coefficients that are stored during the compression stage, this technique could be classified as being mainly a compression algorithm or a level-of-detail technique.

In Figure 3.4 on the following page is an example of a 1D data set, that has been compressed using this technique. The first image shows the original data. The other three images show the reconstructed data, using 50% and 25% of the coefficients and the 5 largest coefficients, respectively.



(a) The original data set

(b) 50% of the coefficients

(c) 25% of the coefficients

(d) The 5 largest coefficients

Figure 3.4: The values of the *Standard & Poors 500* stock market index for the year 2001 [2].

3.3 Multi-resolution data structures

Multi-resolution techniques use specific data structures for storing and/or retrieving the data at several levels of resolution. Depending on the amount of available time or memory, a particular resolution is used, e.g. for visualisation.

An example of this approach is the Time-Space-Partitioning Tree by Shen et al. [80], in which both the temporal and spatial dimensions are stored at every possible (2^n) resolution, using bintree and octree representations, respectively. This technique will be discussed in the next Subsection.

A popular technique for multi-resolution representation of data is the use of wavelets. (See also Section 3.2.) Two examples of data structures using wavelets are presented in Subsections 3.3.2 and 3.3.3.

A final example of a multi-resolution data structure was presented by Pascucci and Frank [51, 52]. They use the Lebesque or Z-order space-filling curve to transform the data to a hierarchical multi-resolution representation. This representation rearranges the data according to the level of resolution, in such a way that subsequent levels of resolution of the data are stored consecutively on disk. On traversal, the lowest level of resolution will be accessed first; increasingly higher levels of resolution will be accessed when required. We have selected this data structure for further study. More details about our work on the Lebesque space-filling curve will be presented in Chapter 6.

3.3.1 Time-Space Partitioning Tree

The Time-Space Partitioning (TSP) Tree was presented by Shen et al. [80] for fast volume rendering of time-varying fields. The TSP Tree can capture both temporal and spatial coherence in a time-dependent data set.

The basic structure of a TSP Tree is a complete octree, which recursively subdivides the entire volume. Each of the nodes of this octree represents a certain subvolume, the size of which depends on the level in the octree. In each of the nodes, however, not only spatial information is stored, but all the temporal information about the subvolume as well. Each node contains a complete binary tree, subdividing the temporal domain. Therefore, each node of this binary time tree represents the subvolume over a certain time range. The data of the root of the binary tree will represent the entire temporal domain, and will contain data values, averaged over the entire time range. The leaf nodes of the binary tree will represent the single time steps and will contain data from a single time step. Again, the level of a node in the octree will determine the size of the subvolume the node represents. The root node of the octree will represent the entire volume, the leaf nodes represent the individual voxels. The level of a node

in the bintree will determine the size of the time span the node represents.

Having this tree structure, the data can be queried at any spatial or temporal resolution, ranging from a single number representing the temporal average of the entire volume, down to an individual voxel at a single time step.

The data that is stored in each of the nodes of the binary trees consists of:

- the mean data value of the entire subvolume within the given time span;
- a measure for the spatial error of the subvolume within this time span;
- a measure for the temporal error of the subvolume within this time span.

This data structure was designed for fast volume rendering. Having only these data values, isosurfacing, for example, is not possible. The error values that are stored at each node are used during tree traversal as stopping criteria. When the error values are below the user-provided error tolerances, the tree will not be traversed any further, and the current mean data value will be used. The spatial error will indicate whether the octree has to be traversed further downward. If not, the current mean data value apparently represents the entire subvolume well enough. The temporal error indicates whether the binary tree has to be traversed any further. If not, the current data value can be reused for all time steps within the given time span.

3.3.2 Wavelet-based Time-Space Partitioning Tree

The Wavelet-based Time-Space Partitioning (WTSP) Tree was recently introduced by Wang and Shen [100]. As the name suggests, the Wavelet-based TSP Tree is an extension of the TSP Tree discussed in the previous Section. It uses a normal TSP Tree as the underlying data structure.

However, in the WTSP Tree, this data structure is combined with a two-stage wavelet transform. First, for each block of volume data from the leaf nodes of the spatial octree, a 3D wavelet transform is computed. This will produce low-pass and high-pass filtered coefficients. The low-pass filtered coefficients from eight adjacent subvolumes will be grouped to form a single block of data, constituting the parent node of the eight subvolumes in the octree hierarchy. Then, the same 3D wavelet transform will be applied recursively on this block and its adjacent subvolumes, until the root node of the octree has been reached. This algorithm will be applied to each time step in the data set. In the next step, the high-pass filtered coefficients from corresponding subvolumes in consecutive time steps will be grouped and transformed using a 1D wavelet transform, resulting in a temporal hierarchy for each subvolume.

The whole process is illustrated in Figure 3.5 on the next page.

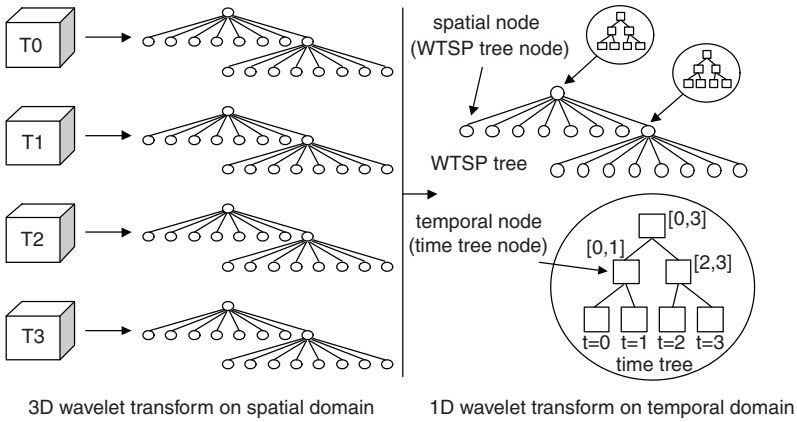
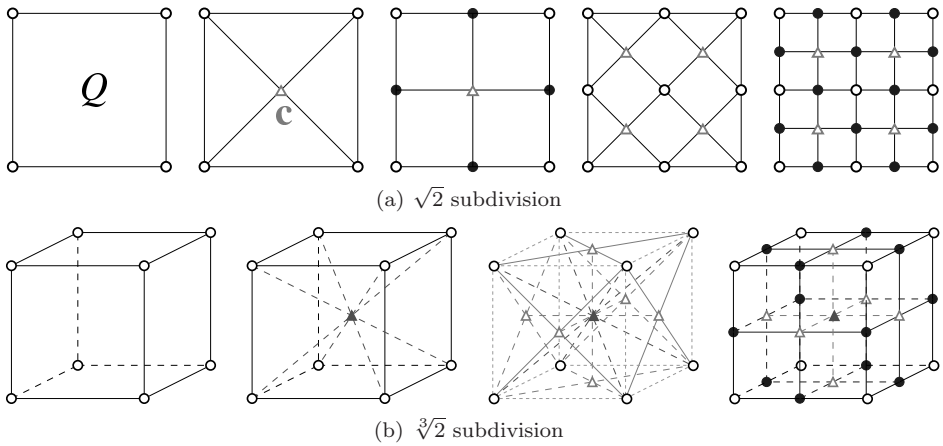


Figure 3.5: An illustration of the construction of the WTSP Tree. The volume is first subdivided into an octree for each time step. For each node in the octree, a 3D wavelet transform is performed. This results in low-pass and high-pass filtered coefficients. The low-pass filtered coefficients are gathered to build up the spatial hierarchy. The high-pass filtered coefficients across all time steps are processed with a 1D wavelet transform to build up the temporal binary tree at each octree node [100].

Further data reduction is performed by using a threshold parameter for the high-pass filtered coefficients: the coefficients are set to zero if they are less than the threshold. Finally, the coefficients resulting from the 1D wavelet transform are further compressed using run-length and Huffman encoding. Again, this is an example of a technique that cannot be uniquely classified. Although discussed here as a multi-resolution data structure, this algorithm makes extensive use of compression techniques.

3.3.3 Wavelet-based multi-resolution with $\sqrt[n]{2}$ subdivision

Linsen et al. recently presented a multi-resolution method that makes use of $\sqrt[n]{2}$ subdivision [40]. Regular subdivision schemes suffer from coarse granularity. For example, in one octree subdivision step, the number of vertices is multiplied by a factor of eight. Using $\sqrt[n]{2}$ subdivision only doubles the number of vertices in each subdivision step, corresponding to a factor of $\sqrt[n]{2}$ in each of the n dimensions. In each step of the subdivision algorithm, an element is split by adding the centroid of the element as a new vertex, connecting the centroid with the old vertices and removing the old edges. Figure 3.6 on the following page shows the first steps of the subdivision algorithm in 2 and 3 dimensions.



(a) $\sqrt{2}$ subdivision

(b) $\sqrt[3]{2}$ subdivision

Figure 3.6: The first steps of the subdivision algorithm in 2 and 3 dimensions [40].

Furthermore, downsampling based on a grid structure in a regular data structure such as an octree introduces aliasing artefacts, possibly resulting in important details being lost. Therefore, Linsen et al. use downsampling filters based on n-variate B-spline wavelets, leading to better approximations of the data at coarser levels.

3.4 Fast access data structures

Fast-access data structures have been specifically designed to perform certain visualisation tasks efficiently. For example, Shen et al. described the Span Space [81] data structure, designed to do fast isosurface extraction. Later, Shen combined this data structure with the Interval Tree [10] for even better performance [79]. This resulted in the Temporal Hierarchical Index Tree, which will be introduced in the next Subsection. As another example, the Shell data structure, by Udupa and Odhner [93] was designed to speed up volume rendering.

3.4.1 Temporal Hierarchical Index Tree

The Temporal Hierarchical Index Tree, or THI Tree, was presented by Shen as a data structure for fast isosurface extraction from time-varying fields [79]. This data structure, which will be discussed in more detail in the following Chapter, is an example of a fast-access data structure. Internally, it uses two other data structures that enable a fast isosurface cell search. As was already mentioned in Section 1.3, this is an advantage as well as a drawback of such methods; it was designed for fast

isosurface extraction, but this is also the only type of visualisation that can be done with the THI Tree.

The basic structure of the THI Tree is a binary time tree. Contrary to the TSP and WTSP Tree, however, this binary tree is not used to create a multi-resolution representation of the data set. In the THI Tree, the purpose of the binary time tree is compression. In each node of the binary tree, only those cells will be stored that are approximately constant for the corresponding time span. This means that long "runs" of constant values — in the time dimension — are stored only once: a kind of run-length encoding. So, this data structure in fact uses a combination of fast access algorithms and a time compression technique. The details of the THI Tree will be discussed in Chapters 4 and 5.

3.5 Discussion

When we have to make a comparison between all these methods and data structures, there are several criteria we can take into account. First, we can look at the potential of the methods for solving the large data handling problem. How well can the algorithms deal with very large time-dependent data sets? Secondly, are the methods suitable for interactive visualisation? A third criterion, especially of importance within this project, is the suitability of an algorithm for isosurfacing. Other criteria, which are more or less related to the first one are, for example, whether or not an algorithm is suitable for out-of-core processing, in what way the time-dimension is treated, and how much data reduction can be achieved.

Without trying to explicitly evaluate these criteria for each of the methods discussed above, we will briefly summarise the results for this comparison.

Multi-resolution approaches in general have a good potential for handling large time-varying data sets, the advantage being that they can adapt the resolution, depending on the amount of available time and/or main memory. "Solving the problem" might be too much to ask, however. The disadvantage of many data structures is that they often increase the total amount of data.

The (W)TSP Tree algorithms have the drawback that they are limited to volume rendering. This (almost inevitably) means, they can hardly be interactive. If not the data retrieval, then the rendering will certainly be the bottleneck. The THI Tree was designed for, and is limited to isosurfacing, but it does this very fast. The other methods discussed here are more general and will allow more than one type of visualisation.

Any method that adopts a multi-resolution or level-of-detail approach can be used out-of-core. The method by Ibarria et al., using the Lorenzo predictor, was designed

having a small memory footprint, specifically for out-of-core application. We have adapted the THI Tree data structure for out-of-core application. See Chapter 5. Other data structures (e.g. the TSP Tree) were designed to be I/O-efficient.

Some algorithms will handle the temporal dimension as if it were just one of the four dimensions. The methods by Linsen et al. and Ibarria et al. will represent a time-varying data set as a regular 4D array. One advantage of such a representation, apart from the aesthetic point-of-view, is the possibility to perform time tracking by 4D isosurface extraction. Other algorithms handle space and time separately. Often, an algorithm will be applied to the individual time steps. In the technique presented by Baldwin et al., this could be the case. The multi-resolution representation of Pascucci and Frank could be applied to either the 4D array, or to the single time steps individually. Because the latter has some advantages, we adopted that approach. In short, the disadvantage of representing the 4D array in multi-resolution, is that changing the spatial resolution will also change the temporal resolution. Also, for this algorithm, the size of the data set has to be the same in all dimensions. In Chapter 6 we will discuss this data representation in detail and also briefly explain these two approaches and their differences. In the (W)TSP Tree data structures the temporal dimension is stored in a binary tree in order to provide a multi-resolution view on the data. Time is stored separately from space, because of the different characteristics of these dimensions. Here, the same argument holds as above. For the 4D representation to work, the temporal and spatial dimensions should always be of the same size and accessed at the same resolution. Furthermore, when data coherence exists in space, but not in time, or in time, but not in space, this coherence cannot be utilised in a 4D-tree, but it can in the time-supplemented octree, that is the TSP Tree [80]. In the THI Tree data structure, finally, the time dimension is used for compression. Time is also stored in a binary tree, but not with the purpose of multi-resolution access. This binary tree is used to represent the temporal coherence in the data.

With regard to the amount of data compression for the various algorithms, this often depends heavily on the data itself. The method by Ibarria et al. could, theoretically, achieve a large data reduction, when the prediction is exact. However, the reduction is only on the basis of delta encoding. The amount of data reduction for the THI Tree algorithm also very much depends on the data. In this case, the amount of temporal coherence in the data is of most importance. Here as well, a large data reduction could theoretically be achieved, if there is very little change in the data over time. On the other hand, the spatial overhead of this data structure, or of structures like the (W)TSP Tree can also be substantial. The method by Baldwin et al. has a flexible amount of data reduction, because during compression the user can provide values for the error tolerance or the desired compression rate. Also the THI Tree algorithm has an error tolerance parameter, with which the user can control the amount of data reduction. Only the multi-resolution representation by Pascucci and Frank is straightforward in this respect. In this algorithm, the data is only reordered; there is

no data reduction, but also no spatial overhead of the data structure.

In the context of this research project, we have further investigated two of the above data structures. First of all, we studied the THI Tree by Shen [79], because it is specifically aimed at isosurfacing, and makes smart use of temporal coherence in the data. This data structure and our extensions to it, are described in detail in Chapters 4 and 5. Secondly, we studied the multi-resolution data structure using the Lebesque space-filling curve by Pascucci and Frank [51, 52]. This is a more flexible data structure, because it is not limited to isosurfacing. Also, because of its multi-resolution representation, it is more scalable to higher-resolution data sets. This method will be further explained in Chapter 6.

CHAPTER 4

Fast time-dependent isosurface extraction and rendering

Benjamin Vrolijk, Charl P. Botha and Frits H. Post

In Alexander Pasko, editor,
Proceedings of the 20th Spring Conference on Computer Graphics
pages 45–54, 2004, ACM Press, ISBN 1-58113-967-5.

Abstract

For the visualisation of time-dependent data sets, interactive isosurface extraction and rendering is desirable. It allows the user to study the development of a surface shape in time, such as a moving front or an evolving object shape. For this purpose, the user must be able to interactively specify an isovalue, and a sequence of isosurfaces must be visualised, starting from any time step, in forward or backward direction in time. In this paper, we describe efficient and tightly coupled techniques for time-dependent isosurface extraction and rendering at interactive frame rates. In preprocessing, we create data structures from a time-dependent data set that allow real-time extraction of all isovalue-spanning cells, achieving rates of several hundreds of frames per second. These isovalued cells are then passed to a fast hardware-assisted direct point rendering algorithm for display, thus avoiding time expensive surface construction by triangulation. This algorithm makes effective use of the available graphics hardware.

4.1 Introduction

Interactive exploration of large, time-dependent data sets is one of the greatest challenges in visualisation today. This is especially true for areas such as flow visualisation, where time-dependent simulations are becoming common practice, and can produce high resolution grid data sets with many thousands of time steps. In spite of this, scientists investigating these large data sets require interactive visualisation techniques with which they can browse through the data in both space and time.

When using a flexible, general-purpose visualisation technique such as isosurface extraction for a time-varying data set, it is desirable to interactively change the isovalue, and watch the development of the surface shape over time. However, extracting and rendering isosurfaces separately for each time step is generally too slow for interactive exploration.

Our approach to this challenge is to use specialised data structures allowing very fast access and data retrieval for answering a specific type of visualisation query, such as in isosurface extraction. We used a number of criteria in choosing such a data structure. First, it should do fast isosurface extraction for any isovalue. Second, it should be suitable for time-dependent data sets. Combining these two, it should be possible to do incremental surface extraction, or to determine the differences between successive time steps. Of course, it should be much faster than straightforward isosurface extraction from every time step. Finally, the results of the extraction should be directly passed to a fast rendering algorithm for display.

We have employed a data structure for fast isosurface extraction from time-dependent data sets [79]. It is specialised, because it does not allow for other types of visualisation, but it is generic in the sense that any isovalue can be extracted from any time step. To make our system achieve interactive frame rates in browsing a data set, we have directly linked the output of our isosurface extraction with a fast, hardware-supported direct rendering algorithm [6], resulting in interactive isosurface extraction and visualisation from time-varying data sets. The direct rendering avoids the time-consuming construction of polygonal surfaces using a Marching Cubes type of algorithm [42]. By combining these two methods, and capitalising on incremental surface extraction, the user can specify an arbitrary isovalue and time step, and the development of the isosurface can be dynamically visualised in forward or backward time direction.

This paper is organised as follows. In Section 4.2, we discuss related work in isosurface extraction techniques from time-dependent data, and suitable rendering techniques to display the isosurface. Then we will explain the data structures we have used in Sections 4.3 and 4.4, and the modified shell rendering algorithm in Section 4.5. Some performance results are given in Section 4.6, and we will reflect on the results and further work in Section 4.7.

4.2 Related work

Most data structures for fast isosurface extraction are based on tree representations. Sutton and Hansen introduced the Temporal Branch-on-Need Tree (T-BON) [89]. This is an extension to the original Branch-on-Need Octree (BONO), described by Wilhelms and Van Gelder [104]. The T-BON is a version for time-dependent data sets, but it does not make use of temporal coherence. The data structure is suitable for fast isosurface extraction.

Shen presents an algorithm for fast volume rendering of time-varying data sets, using a new data structure, called the Time-Space Partition (TSP) Tree [80]. This structure could also be adapted for fast isosurface extraction. The TSP tree is capable of capturing both spatial and temporal coherence in a time-dependent field. Both the spatial and temporal domain are represented hierarchically in the TSP tree: each node of the octree representing space, contains a full bintree representing time. Although this makes multi-resolution access possible for any dimension, it also means a huge storage overhead.

Shen describes another data structure for isosurface extraction from time-varying fields, called the Temporal Hierarchical Index Tree [79]. The idea behind this structure is to store voxels that remain (more or less) constant throughout a certain time span only once for that entire time span. For our purposes, we decided to use and extend the latter. We will describe this structure in more detail in the following Sections.

We have made an implementation of this data structure with optimisations for space efficiency. We have created search routines for retrieving the isosurface-spanning cells for any isovalue and from any time step, and specialised *incremental* search routines that allow an even faster cell search from any time step, given the previous results from another time step.

For visualisation we implemented two different point-based rendering techniques. The first, ShellSplatting, is a hardware-accelerated direct volume rendering method that is based on a combination of splatting [103] and shell rendering [93]. The second is a much faster, but lower quality, point-based volume rendering method that was created specifically for the isosurface extraction documented in this paper. The points are displayed as opaque, flat-shaded polygons that are parallel with the viewing plane. This is an extreme simplification of systems like QSplat [69] and object space EWA surface splatting [63].

Both rendering techniques have been tightly coupled with the extraction technique. The cells that result from the search routines are fed directly into the rendering algorithm, without the need for retrieving the raw data or having to perform interpolation or triangulation. This high level of integration between extraction and rendering is an important advantage of our technique.

4.3 Data structures

Isosurface extraction involves selection of the voxels, or cells, that are intersected by the isosurface, that is, those cells that contain the isovalue. This means that those cells must have some vertices with scalar values lower and some with values higher than the isovalue. To check if a cell is intersected by the isosurface, it is therefore sufficient to store the extreme values of the cell. It is the main idea for this and other data structures, that each cell is stored as an interval $[min_i, max_i]$, and to check if a cell is an isosurface cell, we simply check if the isovalue is contained in that interval.

The data structure we used consists of three elements: a binary tree representing time, and the Span Space and Interval Tree data structures for making an efficient interval search possible. We will discuss each of these structures in the following Sections, before describing the Temporal Hierarchical Index Tree in Section 4.4.

We will use the terms *voxel* and *cell* alternately throughout this paper. Also, in this context, the term *interval* refers to the representation we use for cells or voxels.

4.3.1 Binary Time Tree

An important aspect of the Temporal Hierarchical Index Tree (or THI Tree), is the use of temporal coherence of cells. Instead of storing all the data set's cells for each time step, cells that remain more or less constant (that is, within a certain tolerance) throughout a given time span, are stored only once for that entire time span.

The basic structure of the THI Tree is a Binary Time Tree, dividing the entire range of time steps of the data set recursively into smaller and smaller ranges. The nodes at one level of the binary tree represent a single time step of the data set at a certain temporal resolution. The temporal resolution doubles with each level of the binary tree. See for a simple example Figure 4.1 on the next page. In each node of this binary tree, the cells are stored that remain more or less constant throughout the corresponding time interval. This means that those cells need not be stored anywhere in the tree below the current node. This is the main cause for the possibly large data reduction that can be achieved using this data structure.

The top node of the binary tree represents the entire range of time steps of the data set. The leaf nodes of the tree represent the single time steps at the highest temporal resolution. To retrieve the isosurface cells for a certain time step, the binary tree must be traversed from root to leaf nodes. The cells that are found first, are cells that remain more or less constant throughout the entire time range. The cells that are found in the leaf nodes are those that differ with respect to the neighbouring time steps. Only when the tree has been traversed entirely from root to leaf node, all isosurface cells have been found.

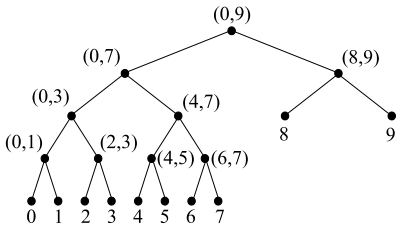


Figure 4.1: An example of a Binary Time Tree for 10 time steps.

We still need a way to classify the variance of cells — we need a way to define "more or less constant" — to determine where the cells should be stored in the Binary Time Tree. Furthermore, we need a way to store a (possibly large) number of cells in each binary tree node efficiently, enabling a quick and efficient search for isosurface cells. Both these problems will be addressed next, when we discuss the Span Space.

4.3.2 Span Space

As stated above, cells are stored in the THI Tree as intervals $[min_i, max_i]$, and isosurface cells are simply those cells for which the interval contains the isovalue. The *Span Space*, as described by Livnat et al. [41], is used to represent intervals $[min_i, max_i]$ as points (min_i, max_i) in 2D. The x-coordinate of a point represents the minimum value, or left extreme, of the interval, and the y-coordinate of the point represents the maximum value, or right extreme of the interval. See Figure 4.2a.

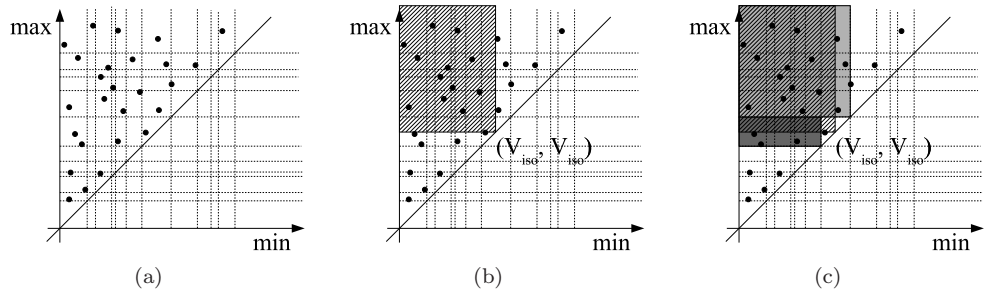


Figure 4.2: (a) Intervals represented as points in Span Space. (b) The intervals spanning a given isovalue V_{iso} are located in the upper left corner from the point (V_{iso}, V_{iso}). (c) The search for intervals spanning a given isovalue V_{iso} is done in three steps, corresponding to three regions in Span Space.

For a time-dependent data set, each cell corresponds to multiple points in Span Space,

one for each time step. The amount of temporal variation of a cell can be quantified by the amount of variation of the corresponding points in Span Space. For this, it is useful to define a grid in the Span Space, for example using a lattice subdivision scheme [81] (see below). As a measure for the temporal variation of a cell, we use the number of grid elements that the corresponding points in Span Space occupy. For example, if all points for a cell during a certain time interval are located within 2×2 lattice elements, we classify the cell as one of low temporal variation for that interval, and therefore, the cell has to be stored only once for that time interval, in the corresponding node of the THI Tree. We use the parameter MaxVariation for this; in Sections 4.4.1 and 4.6 we will discuss the influence of this parameter on the accuracy and size of the THI Tree.

The lattice subdivision scheme used, works as follows. A sorted list is created of all distinct extreme values of all cells from all time steps. From this list, $L + 1$ scalar values are found that divide the list into L equal length sublists. These $L + 1$ scalar values can then be used to draw the $L + 1$ vertical and horizontal lines in Span Space to form the lattice.

The Span Space is not only used for quantifying the amount of temporal variation of the cells, but also for storing the cells in each node of the THI Tree. We store one Span Space per node of the tree. Because non-leaf nodes of the THI Tree represent time spans, instead of single time steps, the cells that are stored in the Span Spaces in these nodes have to be represented by their temporal extremes: a single cell, changing over a number of time steps, corresponds to a number of points in Span Space (one for each time step), but will always be represented by a single point, representing the temporal extreme values.

The points that are stored in Span Space are organised per row of the Span Space. For each row, two lists of points are maintained, one sorted on the minimum value in ascending order, and one sorted on the maximum value in descending order. These lists do not contain the points from the lattice element on the diagonal, because this element requires a min-max search. Instead, these points are stored in a separate data structure, an *Interval Tree* [10]. This structure will be discussed in Section 4.3.3.

When the Span Space needs to be searched for isosurface cells, first, the lattice element $[I, I]$ is located that contains the isovalue V_{iso}, represented by the point (V_{iso}, V_{iso}). See Figure 4.2b.

1. For each Span Space Row $R, R > I$, we search the list that was sorted according to the *minimum* values. We collect the cells from the beginning of the list until the first cell is found with a minimum value greater than the isovalue. Because $R > I$, we know that the maximum values are larger than the isovalue, therefore, all cells found are guaranteed to contain the isovalue.

2. For the Span Space Row I, we search the list that was sorted according to the

maximum values. We collect the cells from the beginning of the list until the first cell is found with a maximum value less than the isovalue. Note that, because we have left out the cells from the diagonal element, all cells in this list have a minimum value less than the isovalue, and therefore, all cells found are guaranteed to contain the isovalue. This is the reason why the cells from the diagonal element are stored in a separate data structure.

3. For the same Span Space Row I, we search this data structure, the Interval Tree, to find the cells from the lattice element $[I, I]$.

In Figure 4.2c, these three cases are illustrated. The first case corresponds to the light grey region. The second case corresponds to the dark grey region, which is the row containing the isovalue. The third case corresponds to the lattice element on the diagonal, for which only the striped part contains isosurface cells; the white parts have either a too large minimum, or a too small maximum value.

4.3.3 Interval Tree

The Interval Tree is a data structure that was proposed by Edelsbrunner [13] to retrieve from a set of intervals those that contain a certain query value. It has an optimal efficiency of $O(\log n)$. We use the Interval Tree to search for intervals (meaning cells) that span a given isovalue [10].

An Interval Tree is created as follows. Given a set $I = \{I_1, \ldots, I_m\}$ of intervals $[a_i, b_i]$, we create a sorted sequence of distinct extremes $X = (x_1, \ldots, x_h)$, that is, each a_i or b_i is equal to some x_j. The Interval Tree consists of a balanced binary tree, whose nodes correspond to values of X, plus two lists of intervals appended to each non-leaf node of the tree. In Figure 4.3 on the facing page is a simple example of a small Interval Tree.

The root of the tree is assigned the "halfway" value $\delta_r = x_{\lceil \frac{h}{2} \rceil}$. The set I is partitioned into three subsets:

- $I_l = \{I_i \in I | b_i < \delta_r\}$; the intervals that are entirely to the left of δ_r;
- $I_r = \{I_i \in I | a_i > \delta_r\}$; the intervals that are entirely to the right of δ_r;
- $I_{\delta_r} = \{I_i \in I | a_i \leq \delta_r \leq b_i\}$; the intervals that contain or overlap δ_r.

The intervals in I_{δ_r} are stored in the root node, arranged into two lists: one containing all intervals sorted according to their left extremes a_i, in ascending order (AL), and one containing all intervals sorted according to their right extremes b_i, in descending order (DR).

The left and right subtrees are defined recursively, by considering the interval sets I_l and I_r, and the sequences $(x_1, \ldots, x_{\lceil \frac{h}{2} \rceil - 1})$ and $(x_{\lceil \frac{h}{2} \rceil + 1}, \ldots, x_h)$, respectively.

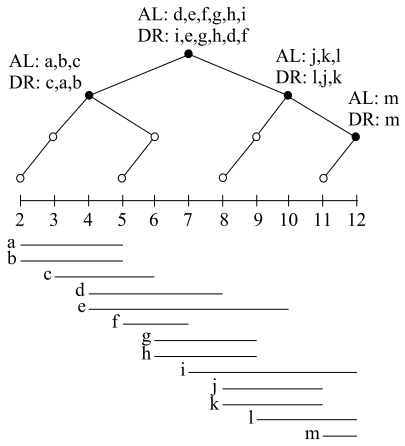


Figure 4.3: An example of a simple Interval Tree for a small number of intervals.

When searching the tree for a given isovalue V, the tree is traversed as follows, starting at the root:

- if $V < \delta_r$ then list AL is scanned until an interval I_i is found such that $a_i > V$; all scanned intervals are returned and the left subtree is traversed recursively;
- if $V > \delta_r$ then list DR is scanned until an interval I_i is found such that $b_i < V$; all scanned intervals are returned and the right subtree is traversed recursively;
- if $V = \delta_r$ then list AL is returned.

4.4 Temporal Hierarchical Index Tree

We now have all the tools to construct the Temporal Hierarchical Index Tree. We do all this in a preprocessing step.

First we classify all cells according to their variance over time, using the Span Space, in order to determine their locations in the Binary Time Tree. For each cell, an interval is determined, which is represented as a single point in Span Space. The variation over time is quantified by the number of grid elements in Span Space that are occupied by the points corresponding to that cell in each time step. Cells with a low temporal variation over a long time span are placed high up in the tree. Note

that the time tree structure is determined *a priori*, only by the number of time steps. Therefore, the time intervals which are represented by each node of the tree, are fixed. Referring to Figure 4.1 on page 64, if a cell remains constant for the time interval $[0, 5]$, for example, it will be stored in the two nodes $[0, 3]$ and $[4, 5]$, because there is no node for the interval $[0, 5]$.

Next, we store all cells for a certain node of the tree in a single Span Space, arranging the cells per row of the Span Space into two lists plus an Interval Tree. We use the same Span Space, meaning the same lattice subdivision, for every Span Space in the THI Tree. The list of cells for a single Span Space is divided into sublists using this lattice subdivision. Each sublist contains the cells for one row of the Span Space. The cells that belong to the lattice element on the diagonal, are stored in an Interval Tree, and removed from the sublist. The remaining cells are stored in two separate lists, the one sorted according to ascending minimum value, the other according to descending maximum value.

4.4.1 Isosurface cell query

The Temporal Hierarchical Index Tree can be queried for any isovalue at any time step. First of all, we determine the Span Space lattice element that contains the isovalue, because all Span Spaces used in the THI Tree use the same subdivision. Next, the tree is traversed from top to bottom, selecting the correct nodes depending on the requested time step. In each node of the tree, the corresponding Span Space is searched, as described above in Section 4.3.2. The cells returned by every search contribute to the final result, which will be complete when the leaf nodes of the THI Tree have been reached. The list of cells we have obtained now contains all cells in the requested time step that span the isovalue, and therefore, all cells that are intersected by the isosurface. However, cells that are found outside the leaf nodes of the THI Tree, are represented by their temporal extreme values, measured over a certain time interval. The fact that these temporal extreme values span the isovalue does not guarantee that the extreme values for the current time step do so too. This means that the resulting list of cells contains a number of false positives.

The number of false positives can be controlled, but a reduction of this number will be at the cost of memory space. There are two parameters to control the accuracy (and therefore the memory space) of the THI Tree. First, the Span Space grid size can be adjusted (the parameter L, we discussed in Section 4.3.2); smaller grid elements result in fewer false positives. Next, another parameter (MaxVariation) defines which cells are considered as "more or less constant" over time. This parameter corresponds to the number of grid elements that a single cell, varying over time, may occupy in Span Space, and still be called constant. Stated otherwise, this parameter defines the maximum allowed variation of a "constant" cell. Increasing this parameter obviously increases the number of false positives, but reduces the memory size of the resulting

THI Tree. Depending on the setting of these two parameters, the number of false positives ranges from about 0.1% with the largest tree size, up to 5% with the smallest tree size in our test application. Using the default settings, we get approximately 0.5% false positives.

4.4.2 Incremental search

The binary tree structure for representing time spans makes it possible to do incremental searching for isosurface cells. Because each node in the tree represents a certain time span, the information that is known in that node can be used for all time steps in that span, that is, for all child nodes of that node. For example, let us assume that a search has been performed for time step 0, and that the resulting isosurface cells are known. When time step 1 is to be searched next, the tree does not need to be searched fully. Instead, the previous result can be used, because all the cells that have been found from the root of the tree down to the node representing time span $[0, 1]$, can be reused. These cells are identical for both time step 0 and time step 1. Only the leaf node representing the single time step 1 must be searched. Next, when time step 2 is to be searched, we need to do a little more "back-tracking", because the last common node for time steps 1 and 2 is the node $[0, 3]$.

This can be implemented fairly easily. The search in each node of the tree returns a number of cells. These cells are appended to a single result vector. For the incremental search to work, we save the number of cells found so far, that is, the size of the result vector, in a single vector of integers. This vector is the only space overhead for the incremental search — at most d integers, where d is the maximum depth of the time tree.

For an incremental search of any time step t_n, we pass the result vector of the previous search, the integer vector $V[d]$ we just described, and the time step t_o of the previous search. Note that these time steps do not have to be consecutive; any two time steps can be used. The binary tree is then traversed from the root to the leaf node representing t_n. In each node N_i (at depth i), we check whether t_n and t_o are in this node's time span. If so, we simply go to the next node, because we can reuse the first $V[i]$ cells from the result vector. If not, we truncate the result vector after $V[i-1]$ cells, because that is the number of cells that t_n and t_o have in common. The rest of the tree must be searched normally. During this search the result vector and the integer vector V have to be kept up-to-date. While only causing a negligible space overhead, this incremental search routine offers a performance gain of a factor 3 in our test application, when we search 50 consecutive time steps incrementally, as opposed to 50 full searches. In Table 4.1 on page 75 the exact numbers are given (under "Speed up") for several different settings of the parameters.

4.5 Point-based rendering

Making use of traditional triangulation and surface rendering techniques for visualisation would almost negate the advantages of the fast isosurface cell extraction. At worst, it would entail that the original data would have to be read from disc for all selected voxels and that surface interpolation would have to be performed with for example the Marching Cubes algorithm [42].

For us the logical answer was to make use of a point-based direct rendering technique. We further optimised our *ShellSplatting* rendering algorithm [6], a combination of shell rendering and splatting, to take advantage of the *a priori* knowledge that the voxels we are dealing with are completely opaque and together constitute an isosurface. ShellSplatting makes use of special data structures that enable very fast implicit space leaping and back-to-front or front-to-back traversal from any viewing angle. This ordering is very important as the technique makes use of Gaussian textured polygons that are composited and scaled by graphics hardware.

The ShellSplatting technique yields high quality renderings of the extracted isosurfaces. However, due to the nature of the data structures used, the voxels have to be ordered in at least the fastest-changing dimension and this slows down the data conversion stage. We wished to provide a second, much higher speed rendering option.

By opting to use flat-shaded rectangular polygons instead of Gaussian-textured ones, the ordering constraint could be ignored. In return, the rendering quality would be slightly lower. In this second method, the polygon that is to be used for rendering the cells is calculated in the same way as for ShellSplatting.

The polygon is constructed to be parallel to the viewing plane. This is correct for the orthogonal projection case. Strictly speaking, in the perspective projection case each rendered polygon should be orthogonal to the viewing ray that intersects it. However, for efficiency reasons, we make use of slightly larger screen-aligned polygons [33]. The polygon is also constructed so that we can perform all rendering in isotropic voxel space and have the graphics hardware perform necessary anisotropic scaling.

To visualise this construction, imagine a three-dimensional ellipsoid bounding a small neighbourhood around a voxel. If we were to project this ellipsoid onto the projection plane and then “flatten” it, i.e. calculate its orthogonally projected outline (an ellipse) on the projection plane, the projected outline would also bound the projected voxel. A rectangle with principal axes identical to those of the projected ellipse, transformed back to the drawing space, is used as the rendering polygon.

Figure 4.4 on the next page illustrates a two-dimensional version of this procedure. In the Figure, however, we also show the transformation from voxel space to world space. This extra transformation is performed so that rendering can be done in the isotropically sampled voxel space, even if the volume has been anisotropically sampled.

Alternatively stated, the anisotropic volume is warped to be isotropic. The voxel-to-model, model-to-world, world-to-view and projection matrices are concatenated in order to form a single transformation matrix $\mathbf{M}$ with which we can move between the projection and voxel spaces.

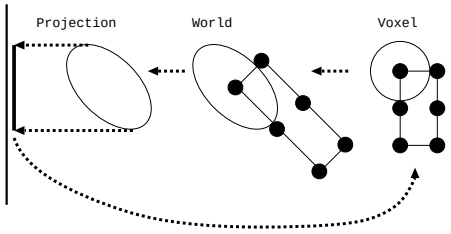


Figure 4.4: Illustration of the calculation of the voxel sphere in voxel space, transformation to world space and projection space and the subsequent "flattening" and transformation back to voxel space [6].

A quadric surface, of which an ellipsoid is an example, can be represented in matrix form as follows:

$$\mathbf{P}^T\mathbf{Q}\mathbf{P} = 0$$

where

$$\mathbf{Q} = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix}$$

contains the coefficients of the implicit function defining the quadric and

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Such a surface can be transformed with a 4×4 homogeneous transformation matrix $\mathbf{M}$ as follows:

$$\mathbf{Q}' = (\mathbf{M}^{-1})^T\mathbf{Q}\mathbf{M}^{-1} \tag{4.1}$$

A voxel bounding sphere in quadric form $\mathbf{Q}$ is constructed in voxel space. Remember that this is identical to constructing a potentially non-spherical bounding *ellipsoid* in world space. In this way anisotropically sampled volumes are elegantly accommodated.

This sphere is transformed to projection space by making use of Equation 4.1. The two-dimensional image of a three-dimensional quadric of the form

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix}$$

as seen from a normalised projective camera is a conic $\mathbf{C}$ described by $\mathbf{C} = c\mathbf{A} - \mathbf{b}\mathbf{b}^T$ [87]. In projection space, $\mathbf{C}$ represents the two-dimensional projection of $\mathbf{Q}$ on the projection plane.

An eigendecomposition $\mathbf{C}\mathbf{X} = \mathbf{X}\lambda$ can be written as

$$\mathbf{C} = (\mathbf{X}^{-1})^T \lambda \mathbf{X}^{-1}$$

which is identical to Equation 4.1. The diagonal matrix λ is a representation of the conic $\mathbf{C}$ in the subspace spanned by the first two eigenvectors in

$$\mathbf{X} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where R and t represent the rotation and translation sub-matrices respectively. The conic's principal axes are collinear with these first two eigenvectors.

In other words, we have the orientation and length of the projected ellipse's principal axes which correspond to the principal axes of a voxel bounding sphere that has been projected from voxel space onto the projection plane. Finally, these axes are transformed back into voxel space with $\mathbf{M}^{-1}$ and used to construct the rectangles that will be used to render the voxels.

The list of cells extracted from the THI Tree is uploaded to the graphics pipeline in arbitrary order as a list of view plane parallel polygons. Because all polygons are non-textured and completely opaque, their ordering is not important. As explained above, scaling is done in hardware, so anisotropic volumes are handled correctly.

Figure 4.5 on the facing page shows a single time step of a sample data set rendered with the ShellSplatter and the fast point-based renderer. The ShellSplatted rendering on the left shows the typical fuzziness often associated with splatting-based rendering methods whilst the fast point-based rendering on the right appears slightly jagged due to the use of flat-shaded quads.

4.6 Results

We have used two data sets for testing the performance of the THI Tree and the renderer. The first is a 96^3 data set of an air bubble rising in water and breaking

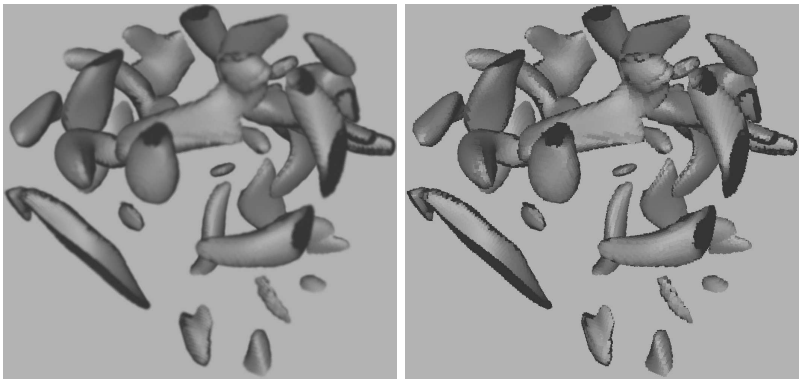


Figure 4.5: Example renderings of a single time step. On the left the high quality ShellSplatting is shown, on the right the faster simple point-based renderer output is shown.

through the surface [1]. This data set consists of 50 time steps ("bubble"). The second data set is obtained from a fluid dynamics simulation, and contains turbulent vortex structures [2]. The size of this data set is $64^3 \times 100$ time steps ("vorticity").

The extraction and rendering performance was measured on a 2.4 GHz Pentium 4 with 1 GB of memory and a 128 MB GeForce 4 Ti4600 graphics card.

4.6.1 THI Tree size

The memory size of the Temporal Hierarchical Index Tree for the 50 time steps of the bubble data set is about 132 Megabytes. The vorticity data set results in a tree size of about 556 Megabytes. This huge difference has to do with the variability of the data and can be illustrated by examining the number of cells in each of the nodes of the tree.

There are two parameters that influence the size of the data structure, and thereby of course, also performance and accuracy.

First, the size of the Span Space can be changed, that is, the number of rows or columns in the Span Space. This affects the number of cells in each row, the number of Interval Trees in the Span Space (one for each row), and the number of cells that has to be stored in each Interval Tree.

However, the total number of cells in the Span Space is not affected, therefore, the

[1] Data courtesy S. P. van der Pijl of Delft University of Technology.
[2] Data courtesy D. Silver and X. Wang of Rutgers University.

memory size of the Span Space will hardly change. Only the vector representing the Span Space boundaries is affected by this parameter, but this vector is stored only once for the entire THI Tree. But if the Span Space contains fewer grid elements, meaning that the grid elements are larger, then cells will sooner be considered constant for a longer time span, and therefore these cells will be stored higher up in the THI Tree, thus reducing the overall size of the data structure. The downside is that more false positives will be found. Accuracy is traded off for memory size.

The same applies to the MaxVariation parameter, which indicates how many grid elements cells may span, and still be considered constant. Thus, without changing the size of each Span Space, we can control the level at which the cells will be stored in the THI Tree. This way we are able to reduce the total memory size of the tree, but at the cost of increasing the number of false positives that will be found.

In Table 4.1 on the next page a few performance characteristics of the Temporal Hierarchical Index Tree are shown. We have used the bubble data set (see Figure 4.6) for determining the influence of the two parameters discussed above. We created THI Trees with 3 variants of each of the two parameters: for the Span Space size, we used values of 32, 64 and 128, and for the MaxVariation we used 1, 2 and 3 grid elements.

Cells in our data structure are represented by a cell id, a minimum and maximum value, and a gradient. We compared the size of the THI Tree to the raw data size, meaning simply the number of time steps $\times$ the number of cells $\times$ the memory size of one cell.

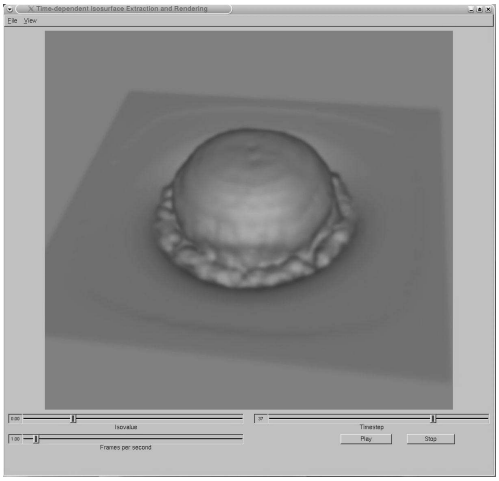


Figure 4.6: A single time step from the bubble data set. The GUI contains sliders for interactively changing the isovalue and the current time step.

Bubble data set, size: $96^3 \times 50$ time steps Raw size of cell data: 981 MB						
SpanSpaceSize	32	64	128	64	64	64
MaxVariation	2	2	2	1	2	3
THI Tree size (MB)	89.2	132.6	193.4	355.3	132.6	96.7
% of raw data	9.1%	13.5%	19.7%	36.2%	13.5%	9.9%
Search (ms)	8.2	8.4	9.0	9.0	8.4	8.1
SearchIncr (ms)	2.0	2.5	2.9	2.7	2.5	2.1
Speed up	4.1	3.4	3.1	3.3	3.4	3.9

Table 4.1: Time and space performance of the THI Tree for different values of the parameters SpanSpaceSize and MaxVariation.

4.6.2 Surface cell extraction

The THI Tree data structure provides a very quick way to search for isosurface cells. In our bubble data set the time for extraction of the isosurface cells for a single time step takes on average approximately 8.4 milliseconds. When we use the incremental search algorithm we can achieve even higher rates: incrementally searching the isosurface cells in 50 consecutive time steps costs about 126 milliseconds. This corresponds to 395 frames per second, or 2.5 milliseconds per frame. In the vorticity data set the average rate of extraction for the 100 time steps is 1186 frames per second.

Referring to Table 4.1, the row "Search (ms)" displays the average extraction time (in milliseconds) of the isosurface cells from a single time step. The next row shows the same, but with the use of our incremental search routine. The last row shows the speed-up of the incremental search, compared to the normal search. In comparison, the average isosurface extraction time for a single time step, using the VTK implementation of Marching Cubes is about 134 milliseconds. This number can be compared to the 8.4 milliseconds for our normal search, or the 2.5 milliseconds for our incremental search.

4.6.3 Rendering performance

We tested the two renderers, both the high quality ShellSplatter and the lower quality fast point-based renderer, with the two data sets. The average frame rates for the total pipeline of extraction and rendering (of 512×512 images) are shown in Table 4.2 on the following page.

Compared to the extraction times, the rendering is the processing bottleneck. The numbers in this Table are rates for combined extraction and rendering, but 80% to 98% of the time is used in the rendering step, depending on the type of renderer used.

Rendering mode	Bubble	Vorticity
High quality	11.1	20.4
Fast	67.4	135.7

Table 4.2: Average rendering frame rates (in fps) for the two data sets, both in high quality and in fast rendering mode, for 512×512 images.

For the rendering, the number of isosurface cells is the most important factor. The average number of isosurface cells extracted from the bubble data set for the chosen isovalue is 16291; for the vorticity data set, the number of isosurface cells is on average 7617 per time step.

4.7 Conclusions and future work

We have described techniques for fast isosurface extraction and direct rendering from time-varying data sets. In a preprocessing step, data structures are generated that allow us to retrieve the isovalue-spanning cells at any time step and for any isovalue with high frame rates. Incremental searching uses temporal coherence to further speed up the extraction process. The extracted cells are rendered directly with a fast point-based rendering technique, displaying a shaded quadrangle at each pixel at high frame rates. No visibility ordering is needed in this case, so the overall speed is not reduced by an intermediate data conversion step. A high quality rendering technique based on ShellSplatting does require visibility ordering, but can still achieve interactive frame rates for a 96^3 data set. In an interactive environment, the fast rendering can be used during interaction, while the high quality technique can be automatically invoked when the input queue is empty. We will integrate this in our VR data exploration system.

With this work, our main contributions are the fast incremental search and the integration of the fast isospanning-cell extraction and rendering stages. We have also attempted to further optimise the search data structures for space efficiency. Even more improvement is possible by using compression techniques, as recently proposed by Bordoloi and Shen [5].

However, in order for our technique to be truly scalable to very large data sets, out-of-core functionality is required. We are currently working on an out-of-core version of the THI Tree in which a limited number of time steps remain in memory, and new time steps are loaded on demand. This completely overcomes the huge memory requirements and makes it possible to visualise very large trees also on systems with only a small amount of memory.

There are two possible sources of error in the display of the isosurfaces that must be

investigated further. Although this did not show up in the test images, the rendering of false positive cells may cause artifacts. Also, the surface normals are stored only once over a time interval that is considered "more or less constant". This also did not have any noticeable effect in the images, but we will analyse the extent of the errors caused.

4.8 Epilogue

This Chapter was presented as a paper at the Spring Conference on Computer Graphics in 2004. As was described in the previous Section, further research is necessary to make the technique truly scalable to very large data sets. In the form in which the technique has been presented in this Chapter, the size of the THI Tree data structure is limited to the amount of main memory available. In order to remove the limitations on the size of the data set, the THI Tree data structure and algorithms have to be adapted for out-of-core application.

We have continued our research on this data structure to do just that. We have created an out-of-core version of the THI Tree and the accompanying algorithms, with which it is possible to handle data sets that are much larger than the size of the computer's main memory. In the following Chapter, these extensions and adaptations will be described in detail.

CHAPTER 5

Interactive out-of-core isosurface visualisation in time-varying data sets

Benjamin Vrolijk, Frits H. Post

Computers and Graphics
vol. 30, no. 2, April 2006, pages 265–276, ISSN 0097-8493.

Abstract

We present a combination of techniques for interactive out-of-core visualisation of isosurfaces from large time-dependent data sets. We make use of an index tree, computed in a preprocessing stage, which effectively captures temporal coherence in the data set. This tree data structure enables fast extraction of all isovalue-spanning cells from any time step and for any isovalue. For very large time-dependent data sets, such as those resulting from CFD simulations, this data structure can easily become too large to fit in main memory. Therefore, we have adapted the generation of the data structure, as well as the data structure itself for out-of-core application. During generation, the data set is spatially divided into several regions, each resulting in a separate tree. For visualisation, the application uses all these trees simultaneously, but will use only part of each of the trees. Only a user-specified time window will be kept in main memory and other parts of the tree will be read and released on demand. Finally, to avoid time-consuming triangulation and surface reconstruction, we have used a hardware-assisted direct point rendering algorithm for displaying the isosurfaces. These combined techniques allow interactive exploration and visualisation of very large time-varying data sets on a normal PC.

5.1 Introduction

One of the greatest challenges in visualisation today, is the interactive exploration of large, time-varying data sets. Especially in areas such as flow visualisation, time-dependent simulations are becoming common practice, and can produce high resolution grid data sets with many thousands of time steps. In spite of the huge size, scientists investigating these data sets need interactive visualisation techniques with which they can browse through the data in both space and time.

Flexible, general-purpose visualisation techniques such as particle tracing, volume rendering, or isosurface extraction are in general not fast enough for time-dependent exploration, or for interactive control of the visualisation parameters. For example, when using isosurface extraction for a time-varying data set, it is desirable to interactively change the isovalue, and watch the development of the surface shape over time. However, extracting and rendering a new isosurface for each time step is generally too slow for interactive exploration.

Our approach to this challenge is to use a specialised data structure allowing very fast access and data retrieval for answering a specific type of visualisation query. We used a number of criteria in choosing such a data structure. First, it should do fast isosurface extraction for any isovalue. Second, it should be suitable for time-dependent data sets. Combining these two, it should be possible to do time-dependent or "incremental" surface extraction, or to determine the differences between successive time steps. This means the data structure should exploit temporal coherence in the data. Of course, it should be much faster than straightforward isosurface extraction from every time step separately. Finally, the results of the extraction should be directly passed to a fast rendering algorithm for display.

We have employed a data structure for fast isosurface extraction from time-dependent data sets [79]. To make our system achieve interactive frame rates in browsing a data set, we have directly linked the output of our isosurface extraction with a fast, hardware-supported direct rendering algorithm [6], resulting in interactive isosurface extraction and visualisation from time-varying data sets. The direct rendering avoids the time-consuming construction of polygonal surfaces using a Marching Cubes-type of algorithm [42]. By combining these two methods, and capitalising on temporal coherence, the user can specify an arbitrary isovalue and time step, and the development of the isosurface can be dynamically visualised in forward or backward time direction (see Figure 5.1 on the next page).

However, the tree data structure used may become too large to fit in main memory. We have overcome the huge memory requirements for creation and use of this data structure. For this, we have adapted the data structure for out-of-core application. We designed and implemented an intelligent paging scheme to enable interactive out-of-core isosurface extraction and rendering on a regular PC.

This paper is organised as follows. In Section 5.2, we discuss related work in isosurface extraction techniques from time-dependent data, suitable rendering techniques to display the isosurface, and out-of-core techniques. Then we will briefly explain the data structures we have used in Section 5.3, together with the out-of-core algorithms and adaptations in Sections 5.4 and 5.5. The results will be discussed in Section 5.6, and we will give our conclusions and directions for future work in Section 5.7.

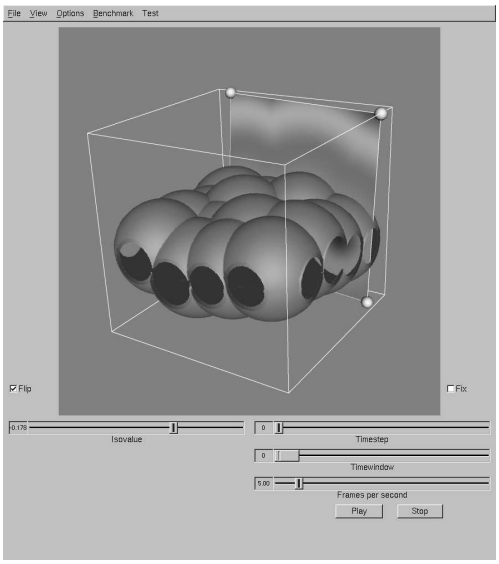


Figure 5.1: A 256^3 data set of air bubbles rising in water. See also colour Figure C.9.

5.2 Related work

Many techniques for fast isosurface extraction are based on tree representations. Sutton and Hansen introduced the Temporal Branch-on-Need Tree (T-BON) [89]. This is an extension to the original Branch-on-Need Octree (BONO), described by Wilhelms and Van Gelder [104]. The T-BON is a version for time-dependent data sets, but it does not make use of temporal coherence. The data structure is suitable for fast isosurface extraction.

Shen presents an algorithm for fast volume rendering of time-varying data sets, using a new data structure, called the Time-Space Partition (TSP) Tree [80]. This structure could also be adapted for fast isosurface extraction. The TSP tree is capable of capturing both spatial and temporal coherence in a time-dependent field. Both

the spatial and temporal domain are represented hierarchically in the TSP tree: each node of the octree representing space, contains a full bintree representing time. Although this allows multi-resolution access in any dimension, it involves a huge storage overhead.

Shen describes another data structure for isosurface extraction from time-varying fields, called the Temporal Hierarchical Index Tree [79]. The idea behind this structure is to store voxels that remain approximately constant throughout a certain time span only once for that entire time span. Within this data structure, two other data structures are used. First, the Span Space representation, as introduced by Livnat et al. [41], is used to store intervals in a two-dimensional space. Second, Interval Trees, described by Cignoni et al. [10], provide an optimal interval search algorithm.

Bordoloi and Shen [5] presented an algorithm for storing intervals more efficiently than in the Span Space, using transform coding.

Recently, Gregorski et al. [15] presented a technique for progressive isosurface extraction with adaptive refinement from compressed, time-dependent data sets. However, they are restricted to playing forward and backward in time. The vertex programming capabilities of modern graphics hardware are used to speed up the surface extraction.

Pascucci also uses the vertex programming capabilities of modern graphics hardware [50]. In his approach, the workload is distributed between the CPU and the graphics card. A tetrahedral decomposition of the domain is used. The application draws one quad per tetrahedron; the vertex program on the graphics card does the interpolation to find the position of the vertices of the isosurface, and computes the normal of the isosurface.

For our purposes, we decided to use and extend the Temporal Hierarchical Index Tree by Shen [79]. We will describe this structure in more detail in the following Sections.

We have made an implementation of this data structure with optimisations for space efficiency. We have created search routines for retrieving the isosurface-spanning cells for any isovalue and from any time step, and specialised *incremental* search routines that allow an even faster cell search from any time step, given the previous results from another time step [96].

We wanted to overcome the huge memory requirements both during creation of the data structure and in interactive visualisation, when the data structure is used. Therefore we have designed a paging scheme for this tree data structure that makes out-of-core tree building and extraction possible for very large data sets. In the application program, this data structure is suitable for paging per time step, unlike for example the TSP tree. Recently, Chiang presented a technique for out-of-core isosurface extraction from time-varying fields [8], which uses as the basic data structure a time tree similar to the one described here. The underlying structures of his technique are however optimised for I/O and out-of-core computation. We have focused on both

fast extraction and rendering and afterwards adapted the data structure and added the paging scheme. We did not try to create I/O-optimal data structures and algorithms; we used this scheme because it suits our data structure. For an overview of out-of-core algorithms for computer graphics and visualisation, we refer to the survey by Silva et al. [82].

For visualisation we developed two different point-based rendering techniques. The first, ShellSplatting, is a hardware-accelerated direct volume rendering method that is based on a combination of splatting [103] and shell rendering [93]. The second is a much faster, but lower quality, point-based volume rendering method that was created specifically for the isosurface extraction documented in this paper. The points are displayed as opaque, flat-shaded polygons that are parallel with the viewing plane. This is an extreme simplification of systems like QSplat [69] and object space EWA surface splatting [63].

Both rendering techniques have been tightly coupled with the extraction technique. The cells that result from the search routines are fed directly into the rendering algorithm, without the need for retrieving the raw data or having to perform interpolation or triangulation. This high level of integration between extraction and rendering is an important advantage of our technique.

5.3 Temporal index tree

Isosurface extraction involves searching the cells that are intersected by the isosurface, which means that they contain the isovalue. Therefore, each of these cells must be enclosed by vertices of which at least one has a scalar values lower and at least one has a scalar value higher than the isovalue. To check if a cell is intersected by the isosurface, it is sufficient to store the extreme values of the cell.

It is the main idea for the data structure we will describe next, that for each cell only an interval $[min_i, max_i]$ is stored. To check if a cell is an isosurface cell, we check if the isovalue is contained in that interval.

We have used and modified the Temporal Hierarchical Index Tree data structure [79]. This data structure makes use of temporal coherence in the data set by storing cells that remain (approximately) constant over a certain time span, only once for that time span.

The basic structure of our index tree is a binary time tree in which each node represents a time range — the root node represents the data set's entire time range, the leaf nodes represent the individual time steps. (See Figure 5.2 on the facing page.) To retrieve the data for a particular time step, we will need to traverse the tree from root to leaf nodes and collect the data found in each node.

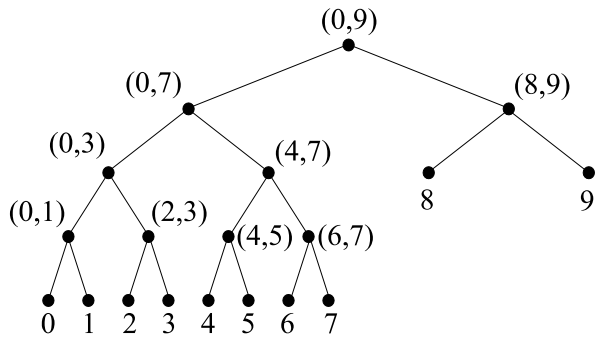


Figure 5.2: An example of a binary time tree for 10 time steps.

In each of the nodes of this binary tree, cells are stored that remain (approximately) constant for that time span. This implies that those cells need not be stored anywhere below that node. Any descendant of a node represents a sub-span of that node's time span, so there is no need to store the cell in the descendant. This is the cause of the potentially large data reduction that can be achieved with this tree structure. Of course, it very much depends on the amount of temporal coherence in a data set. In the worst case, there will be no coherence between successive time steps and all the data will be stored in the leaf nodes of the tree. No data reduction can be achieved in that case. On the other hand, the best possible compression will be achieved when all time steps are similar — all the data will be stored in the root node, and the total amount of data will be equal to the amount of one time step.

5.3.1 Tolerance

We need a tolerance criterion to determine when a cell is considered constant. We have implemented two different criteria, which we call *absolute* and *relative*, from which the user can choose. However, more can be devised and easily implemented. In both cases, the user can specify a tolerance percentage.

In the absolute criterion the tolerance used will be the given percentage of the entire data set's min-max-range. More precisely, we compute the tolerance as follows:

$$T = t \times (\max_g - \min_g), \tag{5.1}$$

where t is the user-provided tolerance percentage; $\max_g$ and $\min_g$ are the data set's global minimum and maximum. This criterion can be used when the data values will be more or less evenly distributed; for every data value, the same tolerance will be used.

In the relative criterion, the given percentage will be taken relative to the current value. For example, a relative tolerance of 1% means that a value of 1.0 may deviate by $\pm$ 0.01, but a value of 10.0 may deviate by $\pm$ 0.1. This type of criterion can be used if the data can be expected to be distributed around a certain value, or if one data value is likely to be picked for the isovalue.

The value of the tolerance parameter influences the construction of an index tree. The results (the size of the data structure, the accuracy of the data structure, and with that the accuracy of the rendering, and the speed of the isosurface cell search) strongly depend on the value of this parameter. But even if the connection between the parameter and the results is intuitively clear, it is not obvious to state the best possible value.

The parameter determines the amount of variance a cell is allowed to have over time and yet be called constant. Obviously, this value should not be too large, otherwise important variations will not be apparent in the results. The value should also not be too low, because then little or no temporal coherence will be found, which would undermine the whole purpose of the binary time tree.

On the other hand, the amount of temporal coherence is a property of the data set. In a very "turbulent" data set, there will be hardly any coherence, but in a "stable" data set, there will be very much.

Ideally, we would like to quantify the overall amount of coherence in a data set and automatically determine the best tolerance parameter for that amount.

Even if we could do this, the question remains what is best. There are several criteria to choose from. For example, the best case with respect to the size of the data structure would be to choose a very high tolerance. This will cause many cells to be called constant and therefore result in all cells being stored in the root node of the binary tree. We get the best possible compression, but the worst possible accuracy. On the other hand, the best case with respect to accuracy would be to have the tolerance set to 0. This will result in most of the cells being stored in the leaf nodes. The compression will be almost 0 and the speed will be very low.

For our purposes, the optimal case will be somewhere in between, having a good amount of compression, but also still a good accuracy. In fact, the highest search speed for a single time step will be obtained when all tree nodes contain about the same number of cells. This is what we will aim for.

5.3.2 Index tree building

When creating an index tree, we start by building the structure of the binary time tree. Note that this structure is determined *a priori*, only by the number of time steps. Therefore, the time ranges which are represented by each node of the tree are

fixed.

From the structure of this tree, and the possible time ranges that can be stored in each node of the tree, we can start classifying the cells in the data set. For each cell, we generate a time-vector v containing the cell's values for each time step. Given this 0D time-dependent data set, (being the temporal evolution of one cell) we start traversing the binary time tree. Referring to Figure 5.2 on page 85, we start at the root node, with time range $[0, 9]$, and check if the current cell remains (approximately) constant for this range. For this we need the cell's time-vector $v[0]$ to $v[9]$. If the cell satisfies the tolerance criterion for this time range, we store the cell in the current binary tree node. If not, we recursively descend the tree and check the child nodes $[0, 7]$ and $[8, 9]$.

To determine if a cell remains constant over a certain time range, we compute its *temporal* extreme values. Because a cell itself contains a minimum and a maximum value, we compute temporal extremes for both the minimum and the maximum. These will be treated similarly. Let's call the minimum L, for *left extreme*, and the maximum R, for *right extreme*. Then, using the absolute criterion, we would check if

$$\begin{aligned} (\overset{t}{\max} L_i - \overset{t}{\min} L_i) &< T \\ (\overset{t}{\max} R_i - \overset{t}{\min} R_i) &< T, \end{aligned}$$

where $\overset{t}{\max}$ and $\overset{t}{\min}$ mean the temporal extremes, and L_i and R_i mean the two spatial extreme values stored in cell i.

For the relative criterion, we would check if

$$\begin{aligned} (\overset{t}{\max} L_i - \overset{t}{\min} L_i) &< T \cdot \| \overset{t}{\min} L_i \| \\ (\overset{t}{\max} R_i - \overset{t}{\min} R_i) &< T \cdot \| \overset{t}{\min} R_i \|. \end{aligned}$$

Note that if the cell is stored, for example, in node $[8, 9]$, meaning that the above criteria return true, its value is called constant for time steps 8 and 9, and there is no need to store the cell in either of the nodes [8] and [9].

Note also that a cell that remains constant for the time range $[0, 5]$, for example, will be stored in the two nodes $[0, 3]$ and $[4, 5]$, because there is no node for the range $[0, 5]$. Moreover, we wouldn't even know the cell remains constant for that time range, because that range will never be checked. Only the ranges that are represented by binary tree nodes will be checked.

If a cell is stored in a node of the binary time tree, we store the cell's id and extreme values. For non-leaf nodes, representing a time range instead of a single time step, this means we have to store the *temporal* extreme values of the cell. For rendering, we also need a normal (or normalised gradient) for each cell. Non-leaf nodes represent

a time range, meaning there are several normals to choose from. Currently, we pick one, for example the middle one; an average normal might be better, although we have not noticed any artefacts with our method.

After all cells have been classified and stored in the binary time tree, they will be reorganised per tree node. In each of the tree nodes, we have maintained a vector of cells. For efficient interval searching, these cells will be rearranged into an *Interval Tree* [10]. This is where we deviate from the original data structure as described by Shen [79]. He uses another data structure, called the *Span Space*, in which intervals are stored as 2D points. We have also implemented this data structure, but eventually rejected it, the biggest disadvantage being that it is very unintuitive. Furthermore, the Span Space in combination with the Interval Trees, as in the original article, does not result in much higher search speed or more efficient storage, than just using Interval Trees.

In our structure, one interval tree will be created in each binary tree node. This structure and its construction are fairly straightforward and described in detail by Cignoni et al. [10] and in our previous work [96]. Therefore, we will not discuss it any further in this article.

5.4 Out-of-core tree building

During the creation of the index tree, we have to iterate through all cells in the data set, determine a time-vector for each cell and store each cell in the right node(s) in the index tree. We need a time-vector for each cell, because we want to determine the time spans in which the cell remains constant. This poses a number of problems.

5.4.1 XYT files

From a practical point of view, the usual way in which time-dependent data sets are stored, is not very well-suited for this purpose. Normally, for each time step a field is stored in a separate file, so the same grid point in different time steps can be located at the same offset in different files. When we need to construct a time-vector for a certain cell, we have to open and search in all files. Either we have to keep all files opened simultaneously, or open and close all files for each cell.

We decided to transpose the entire data set in preprocessing. Instead of storing (x, y, z) data in each file and a separate file for each time step, we transformed the data set to files with (x, y, t) data and with each file representing a different z. Here we assume a regular, Cartesian grid. Of course, other subdivisions are possible, as long as the temporal data for one grid point can be obtained from a single file.

5.4.2 Multiple trees

There is another obstacle when constructing the index tree. Each cell's time-vector is split into a number of time ranges over which the cell remains constant. Then, the cell is stored in the index tree nodes corresponding to those time ranges. In each of these nodes, the cell is appended to a (potentially very large) list of cells for that node. Because each cell may be stored in a number of index tree nodes, it is essential, for efficiency reasons, that we should try to keep the entire index tree in main memory during construction.

Of course, it could be possible to store each index tree node separately on disk during construction, but that would involve a lot of extra disk I/O for every single cell.

Instead we decided to keep as much as possible in main memory, but split the entire data set into several trees. Because of the file layout just described, we decided to split the data set in the z direction. For example, for a 256^3 data set, we have 256 files, each containing (x, y, t) information for a different z value. We could then create 16 index trees, each for a layer of 16 z slices thick. Or 32 trees of 8 slices, or 4 trees of 64 slices. Each of these trees separately can be kept in main memory, so there is no unnecessary disk I/O during construction. When a layer has been completed, the current index tree can be written to disk and cleared from memory, and the next z slice will start a new index tree.

This approach does not give any noticeable space overhead compared to using a single tree. The total disk space needed for all trees is equal to the space needed for a single full tree. In the application program almost no extra processing is required to use multiple trees compared to using a single large tree. In fact, the application just iterates over the array of trees and performs the same function for each individual tree. For an isosurface query, the cells returned by the queries on the individual trees are then concatenated.

Another advantage of using multiple trees, is that we can easily add trees that represent other quantities, meaning we can render several isosurfaces simultaneously. As an extension to this concept, we can also reuse a single tree, enabling us to render several isosurfaces with different isovalues from a single quantity. To the program, these concepts are almost identical. In all cases, we iterate over the array of trees, and for each tree we perform an isosurface cell search. In one case, the isovalues will be the same for each tree, but every tree represents another piece of the data set. In another case, the trees will actually represent the same data, but we will search for different isovalues, and the last case is that we have different trees, each representing another quantity, and each having its own isovalue.

A fourth case is possible, when we have trees representing the same data set and the same quantity, but a different time range. Of course, this is sub-optimal. We should always try to use the data set's entire time range, in order to find the most temporal

coherence. If, somehow, it would not be possible to create index trees for the entire time range, for example because the simulation is still running, it is possible in this way to visualise a time range using several trees. The time ranges for the individual trees will be joined to form one large time range. The difference with the other cases is that at any time only a single index tree will match the current time step, whereas in the other cases all index trees represent the entire time range and thus all trees will always match the current time step. Again, not using the entire time range is sub-optimal. For example, if a data set consisting of 100 time steps is divided into 10 index trees of 10 time steps, it is not possible to find any temporal coherence with a length of more than 10 time steps. This will obviously have a negative impact on the compression ratio.

5.5 Out-of-core visualisation

5.5.1 Time window

During visualisation, we may have a different memory problem. Although we can split the data set into layers during construction of the index tree, we cannot do this during visualisation, as we would like to see the data set's entire spatial extent at once. Therefore, we will have to read all constructed index trees simultaneously. However, we do not need to have all time steps in memory at the same time. We created an intelligent paging scheme that allows us to read only a limited number of consecutive time steps from the index trees into main memory. The structure of the index tree enables us to incrementally read new time steps and remove old time steps from memory. This is the basic idea of our sliding time window concept. We assume this corresponds very well to the way scientists will browse through the data set. We assume the user will normally play through the time steps coherently, either forward or backward. This corresponds to our incremental reading of new time steps. Sometimes, the user may browse a few frames backward and forward, within the time window. If a large jump in time is requested, this will cause a delay, because a completely new time window will have to be read. We assume this is a minor problem, because random jumps in time will probably not happen very often.

As an example, let's take the same binary time tree as in Figure 5.2, representing a data set with 10 time steps. Assume we have a time window containing the 5 time steps $[3, 7]$. In Figure 5.3 on the facing page we have depicted which nodes of the tree will be in main memory. If the time window is shifted one time step to the right, the nodes $[8, 9]$ and $[8]$ will have to be read from disk. The node $[0, 9]$ does not have to be read from disk because it is already in main memory. In fact, the root node will always be in main memory, because it is needed for all time steps. While we need to read extra tree nodes from disk on the one side, we can remove nodes from memory

on the other side: time step 3 is no longer needed, therefore we can delete the nodes $[0,3]$, $[2,3]$ and $[3]$ from main memory.

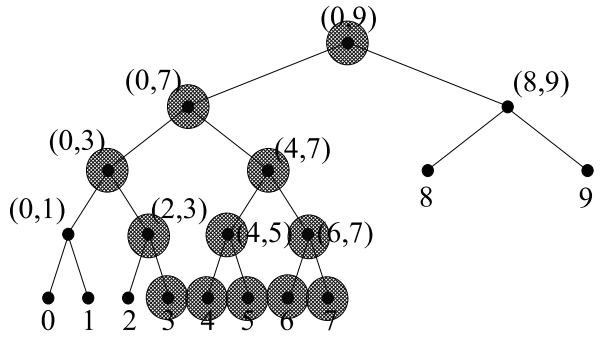


Figure 5.3: A binary time tree with a time window $[3,7]$. Only the coloured nodes will be kept in main memory.

Knowing this, we can define the memory requirements for our algorithm. Evidently, the minimal amount of memory needed is just the space that is needed to store a single time step.

5.5.2 Adaptations to the data structure

To make the time window possible, a number of changes have been made to the index tree data structure. The skeleton of the binary time tree will always be kept in main memory. This means that the tree *structure* will always be available, together with the time span information in each node. Using this structure, we can easily identify which tree nodes have to be traversed to obtain the data for a certain time step. Only the cell data, which is stored in the interval trees in each of the nodes of the index tree, will be eligible for paging to and from disk.

Each index tree is stored as one large binary file on disk. When we have to read the interval tree for a specific index tree node from disk, we need the file offset and the number of bytes to read from our disk file. These two numbers are stored with each node of the index tree.

In fact, the number of bytes is not necessary for reading an interval tree from disk. The entire index tree, but also the individual interval trees will be reconstructed in memory on the fly, while reading from disk. Therefore, we do not have to know in advance the number of bytes to read. However, the number *is* necessary when the interval tree is *not* read; when only the structure of the index tree is read, without the data in the interval trees, it is necessary to know how many bytes to skip to find

the file offset for the next tree node to read.

This way, the *structure* of the index tree can be read entirely from disk, without reading any *data*. The tree then occupies only a few hundred bytes in memory. Next, whenever a time step is requested, the appropriate tree nodes will be read from disk.

When a tree node has to be freed, the interval tree for that node is simply removed from memory.

Of course, we must keep track of which tree nodes currently are in main memory. To this end, we have added a single boolean variable to each tree node. We say a tree node is in memory if the interval tree for that tree node is in memory. Again, the index tree *structure* remains in memory at all times, and therefore, all index tree nodes also exist in memory permanently. Only the interval tree in an index tree node can be paged in and out of memory.

5.5.3 GUI feedback

To provide the user with feedback about the time window, we have designed a GUI element to show a bar from the first to the last time step of the window, with an indicator at the current time step. (See Figure 5.4.)

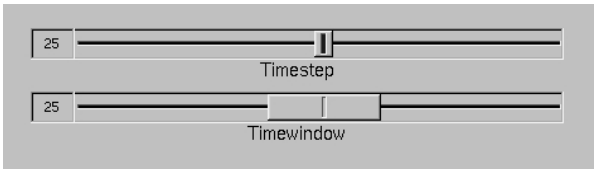


Figure 5.4: The GUI element that shows the time window and current time step.

Because reading new time steps will certainly be slower than visualising them, the visualisation will, in the end, catch up with the last time step of the time window. This is of course dependent on the size of the time window and on the frame rate of the player. It can happen that the user will notice a delay. Therefore it is desirable for the user to get feedback. He will see that the visualisation is catching up with the reading of new time steps and be prepared that he will have to wait. Or, he could slow down the visualisation by lowering the frame rate. Finally, he could also increase the time window if memory size allows this.

5.5.4 Multi-threading

In order to let the visualisation run independently from the reading of time steps from disk, to prevent unacceptable delays, we have decided to use a multi-threaded design for our program. The main thread of the program is concerned with the visualisation. When a certain time step is selected by the user this thread ensures that the part of the index tree containing that time step is resident in memory. If necessary, it will read the corresponding tree nodes from disk. Next, it will perform an isosurface cell search and visualise the result. In the mean time, the second thread is awakened and this thread will start reading new time steps, from the current time step in both time directions, until the requested number of time steps (specified as the window size) has been read in from disk. This could take some time, especially if all time steps in the time window have to be read, but as it happens in a separate thread, the user might not notice anything, while he is investigating the current time step. Of course, when the user simply plays through the data set, only one time step will have to be read at a time, in which case interactive browsing is quite feasible. A third thread has been designed to perform the task of cleaning up unused time steps. This thread will remove all nodes in the tree that are not needed for the current time window.

These last two threads both consist of an infinite loop in which they are suspended while waiting for a signal. The main thread broadcasts the signals whenever a new time step is selected.

5.5.5 Point-based direct rendering

After extracting the cells intersected by the isosurface it would be possible to construct a polygonal mesh for each frame and visualise this using polygon rendering. However, this would take away the advantage of the fast access data structure, as the original data would have to be read from disk in order to perform surface reconstruction using for example the Marching Cubes algorithm [42].

To avoid this, we have used a point-based direct rendering algorithm [96]. We further optimised our *ShellSplatting* rendering algorithm [6], a combination of shell rendering and splatting, to take advantage of the *a priori* knowledge that the voxels we are dealing with are completely opaque and together constitute an isosurface. ShellSplatting makes use of special data structures that enable fast implicit space leaping and back-to-front or front-to-back traversal from any viewing angle. This ordering is very important as the technique makes use of Gaussian textured polygons that are composited and scaled by graphics hardware.

The ShellSplatting technique yields high quality renderings of the extracted isosurfaces. However, due to the nature of the data structures used, the voxels have to be ordered in at least the fastest-changing dimension and this slows down the data

Data set	Bubbles		Clouds	
Resolution	$256 \times 256 \times 256$		$128 \times 128 \times 80$	
# Time steps	39		600	
Raw data size	4 992 MB		3 000 MB	
# index trees	16	8	6	8
xy-resolution	256×256	256×256	128×128	128×128
z-resolution	16	32	80	10
# Time steps	39	39	100	600
Total size	3 170 MB	1 630 MB	824 MB	750 MB

Table 5.1: Details of the two data sets and of the four generated index trees.

conversion stage. We wished to provide a second, much higher speed rendering option.

By opting to use flat-shaded rectangular polygons instead of Gaussian-textured ones, the ordering constraint could be ignored. In return, the rendering quality would be slightly lower. In this second method, the polygon that is to be used for rendering the cells is calculated in the same way as for ShellSplatting.

The polygon is constructed to be parallel to the viewing plane. This is correct for parallel projection. Strictly speaking, in the perspective projection case each rendered polygon should be perpendicular to the viewing ray that intersects it. However, for efficiency reasons, we make use of slightly larger screen-aligned polygons [33]. The details of the construction are described in our previous work [96].

5.6 Results

We have tested our application on two large data sets. The first data set is of a multi-phase flow simulation of a number of air bubbles rising in water. Five double-precision floating point values are computed per grid point: the pressure, the level set value and the three components of the velocity. We use only one scalar to create the index tree, being the level set value; this leaves us with 128 MB of data per time step. See Figure 5.5 on the facing page.

Another data set we used is of a Large Eddy Simulation of cumulus clouds, with one vector and three scalar quantities: the air velocity vector, meteorological temperature, liquid water and total water. For the creation of an index tree, we only used the temperature. See Figure 5.5.

For each of these data sets, we created two sets of index trees; the details are in Table 5.1.

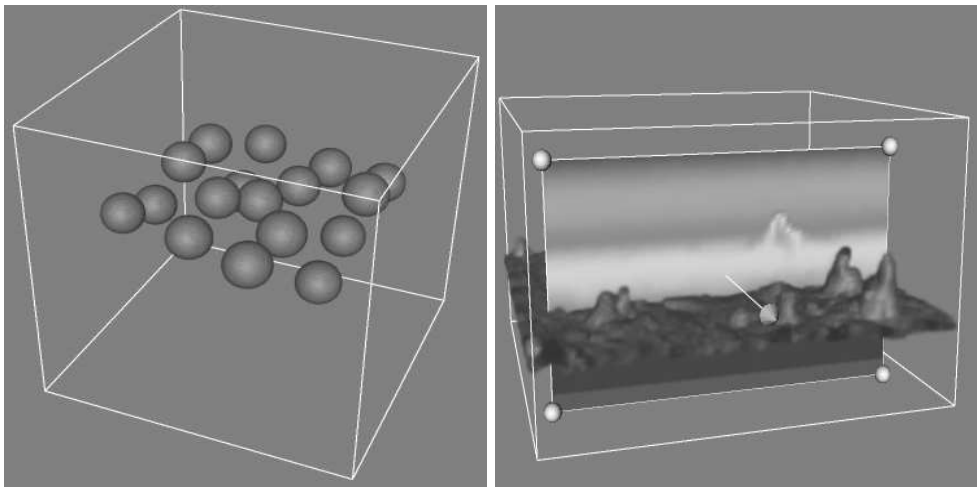


Figure 5.5: Scenes from the two data sets. On the left is the bubble data set, on the right is the cloud data set. See also colour Figure C.10.

5.6.1 Benchmarks

For two of the data sets, we ran a couple of benchmarks. First we ran a rendering benchmark, both with the ShellSplat Renderer and the Fast Point-based Renderer, for different isovalues, meaning different numbers of cells to render. In the other two benchmarks we measured the speed at which we could play through the data set. This involves both extraction and rendering for each time step. This was done for a (worst case) time window of 1, meaning that each time step has to be read from disk before extraction can be done, and for a very large time window. In the latter case, all data is kept in main memory and no data transfers from disk are needed. This is done to test the speed of the extraction algorithm. When we use the ShellSplat Renderer, a sorting step is needed for each time step. To see the influence of this sorting, we have performed the last benchmark with both renderers.

We ran the benchmarks on a modern computer with an Intel Pentium 4 processor, running at 3.0 GHz, and 1 GB of main memory. The graphics card is a NVidia Quadro FX 1300 with 128 MB of memory on a PCI Express graphics bus.

The results of the rendering benchmarks are shown in Figure 5.6 on the next page. It is clear that interactive rendering is possible with the Fast Point-based Renderer, even for over 400,000 cells. Also the ShellSplat Renderer can achieve interactive frame rates up to about 100,000 cells. Because of the texturing and compositing, the ShellSplat Renderer is much slower than the Fast Point-based Renderer.

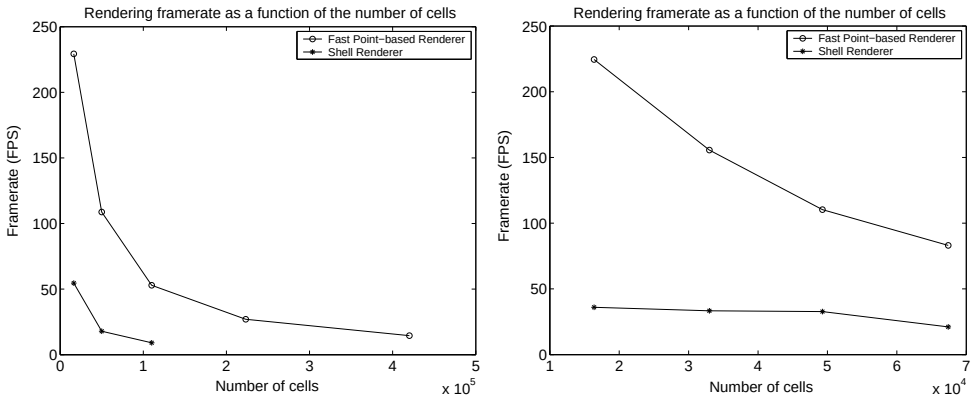


Figure 5.6: The results of the rendering benchmarks. On the left is the bubble data set, on the right is the cloud data set.

Next, we timed at which rate we could play through the entire data set. This involves extraction and rendering for every time step, using the same isovalue. With a time window of 1, only a small amount of main memory is needed, but for every frame, we have to read a new time step from disk into main memory and delete the previous time step from memory. The speed is therefore very much dependent on the amount of data that is to be read per time step. The cloud data set, consisting of 600 time steps, occupies a total of 750 MB on disk, or on average 1.25 MB per time step. We can play through the entire range of 600 time steps at an average rate of 7.8 to 9.5 frames per second, depending on the number of cells to render.

The bubble data set, on the other hand, with only 39 time steps and occupying 1.6 GB on disk, has an average of almost 42 MB per time step. Playing this data set with a time window of 1 is not really interactive, with an average frame rate of about 0.46 fps.

However, if there is more memory available, it should obviously be used. Therefore we also tested the speed at which we could play through the data within a large time window. We used a fixed time window which could be completely stored in main memory; no disk transfers were needed whatsoever. Because the rendering again depends on the number of cells, we ran the benchmarks with different isovalues. The results are shown in Figure 5.7 on the facing page.

Extraction of the isovalue-spanning cells can be done extremely fast. Rendering is also very fast, as long as we don't need the sorting step to create the shell data structure for the ShellSplat Renderer. Construction of this data structure takes so much time that it is not really suitable for interactive use. Once you have made the shell data structure, it is suitable for interactive rendering, but every time you change the iso-

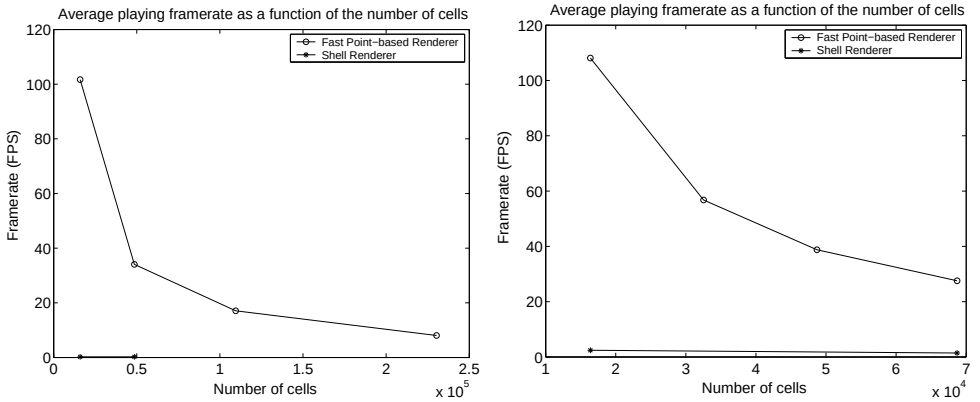


Figure 5.7: The results of the play benchmarks. Playing involves extraction and rendering through (part of) the time range. On the left is the bubble data set, on the right is the cloud data set.

value or the time step, the shell structure has to be regenerated. The recommended use would therefore be to switch to the Fast Point-based Renderer when browsing through time or searching an interesting isovalue. When a particular isosurface in a certain time step has been found and needs to be explored, the ShellSplat Renderer can very well be used interactively.

5.7 Conclusions and Future work

We have presented a combination of techniques to allow interactive isosurface extraction and visualisation from large time-dependent data sets. In preprocessing we create a tree data structure that is designed for fast extraction of all isovalue-spanning cells for any isovalue and from any time step. This data structure makes effective use of temporal coherence in the data by storing values that remain approximately constant over a time range only once for that time range. The cells that are extracted can be quickly rendered using a hardware-assisted direct rendering algorithm. For this direct rendering no interpolation and triangulation for surface reconstruction is needed.

The data structure is in principle limited to isosurface extraction, however, as was shown in Figure 5.5 on page 95, it is possible to make an approximate reconstruction of the original data, providing the ability to perform other algorithms such as slicing.

We have overcome the huge memory requirements for creation of the data structure by spatially sub-dividing the data set and building a separate tree for each subspace. The memory requirements are therefore equal to the amount needed to hold a single

tree.

During visualisation the separate trees can be combined to reconstruct the entire spatial domain. To overcome the memory requirements for this stage, we have designed and implemented a paging scheme, based on a time window paradigm, that will keep only those parts of the trees in memory that are required to visualise the time steps within a user-specified time window. Paging is done per tree node. Each node will be kept in main memory just as long as is needed. The memory requirements for visualisation are therefore equal to the amount of memory needed to hold the time window, which can be as little as one time step.

Within the time window, interactive frame rates can be achieved, for the entire pipeline of extraction and rendering, both for varying isovalue and varying time step. Outside the time window, paging will slow down the frame rates. The amount of data to be transferred from disk will determine the speed. This amount depends not only on the size of the data set but also on the amount of temporal coherence.

Optimisations might be possible by making the data structure more I/O-efficient. This data structure was designed to do fast isosurface cell extraction and later adapted for out-of-core functionality. It can perhaps be optimised for I/O, however, that will be at the cost of in-core performance.

CHAPTER 6

Multi-resolution data representation using hierarchical indexing

6.1 Introduction

In the previous Chapter a fast-access data structure was discussed. This particular data structure was designed for fast isosurface cell search. Although the results for isosurfacing are impressive, the limitations of the data structure are a major drawback. The interval tree structure that is used, is designed for *and limited to* isosurface cell search. The original data is no longer used, therefore it is not possible to combine the isosurface with another type of visualisation. It is possible, although very inefficient, to do an approximate reconstruction of the data from the interval tree, with which, for example, slicing can be done, as was shown in Figures 5.1 and 5.5.

Following the classification of approaches that was introduced in Section 1.2, this Chapter will discuss a multi-resolution data structure. Such a data structure will provide access to the original data at several levels of resolution. Because the original data is used, there is no restriction in the possible types of visualisation that can be performed. However, the multi-resolution access means that when the available time or memory is limited, a lower resolution version of the data will be used, instead of the complete data set.

The data structure that will be introduced in the following was originally presented by Pascucci and Frank [51, 52]. The idea behind their method is a reordering of

the data in such a way that data points from a certain level of resolution are stored coherently on disk. Furthermore, the subsequent levels of resolution will be stored consecutively on disk. This means that low resolution data will be read first, and as time and available memory allow, higher resolution data can be read. In all cases, disk accesses are coherent, because data for each resolution is stored coherently, and therefore cache-friendly. Note that the data is only reordered, meaning that the lower resolutions are merely subsampled, not downsampled versions of the original data. See also Subsection 6.5.1.

In this Chapter, the concept of reordering the data for multi-resolution access will be discussed first. Then, a number of design issues will be presented, that have to be resolved before implementation can commence. After that, in Section 6.4, some results will be given that were achieved with our implementation. Finally, in Section 6.5, we will discuss this implementation and these results and present some ideas for extensions of the data structure.

6.2 Data order

A data set is usually stored in scan line order. The data points are only stored coherently along one dimension, at the highest resolution. Points that belong to the same level of resolution will be scattered throughout the entire data set. To retrieve the data points in a 3D subvolume at a single level of resolution, disk access will therefore be incoherent and inefficient.

By reordering the data in a smart way, this inefficiency can be reduced. The method presented here makes use of the Lebesque or Z-order space-filling curve to transform the data into a hierarchical multi-resolution representation. First, the three-dimensional index of each data point is converted into a one-dimensional index along the space-filling curve. Next, this one-dimensional index is converted to a hierarchical index in such a way that the data points are sorted according to their level of resolution.

In Figure 6.1 on the facing page the principle of the Lebesque space-filling curve is illustrated. The basic shape of the space-filling curve in two dimensions is a Z-shape connecting four data points. In the next level each of these points is replaced by a Z-shape of half the size of the previous one. This is repeated recursively. In three dimensions the basic shape is a pair of Z-shapes, connecting eight data points.

The conversion from an n-dimensional index to the one-dimensional index along the space-filling curve can be carried out by interleaving the bits of (the binary representations of) the n indices. Knowing this, it is straightforward to show that the level of resolution of a certain data point can be easily computed from the binary representation of its one-dimensional index. Suppose level k is the highest resolution. All data

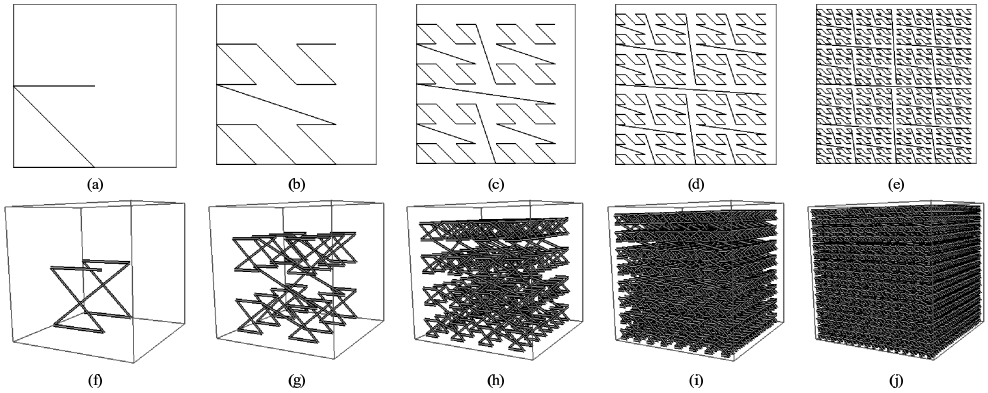


Figure 6.1: The Lebesque space-filling curve. The first five levels of resolution in two dimensions (a–e) and in three dimensions (f–j) [51].

points belong to level k. Points with all *even* indices belong to the second-highest or half resolution (level $k-1$). Points with all indices a multiple of four belong to the quarter resolution (level $k-2$), et cetera. But if all indices are even, the binary representations of these indices will all end with a single zero. Because of the interleaving of the indices, the binary representation of the one-dimensional index along the space-filling curve will end with n zeros. In general, if a data point is part of resolution level $k-h$, the binary representation of the indices of the point will all end with h zeros, and the binary representation of the one-dimensional index will end with nh zeros.

Figure 6.2 on the next page shows an example of the construction of the hierarchical index. On the left, the normal scan line order for a data set is used. It can be easily seen that data points from the same level of resolution are stored irregularly throughout the data set. On the right, the Z-order index is used. Here, data points from the same level of resolution are distributed evenly throughout the data set. This suggests that computation of the hierarchical index can be performed more easily using the Lebesque or Z-order space-filling curve.

The hierarchical index can be computed very easily having the Z-order index. The trailing number of zeros (nh) of the binary representation of the Z-order index gives us the level of resolution i:

$$i = k - h \tag{6.1}$$

where k is the highest level of resolution. The hierarchical index is:

$$I = 2^{n(i-1)} + \left\lfloor \frac{j}{2^{nh}} \right\rfloor - \left\lfloor \frac{j}{2^{n(h+1)}} \right\rfloor - 1 \tag{6.2}$$

where j is the Z-order index.

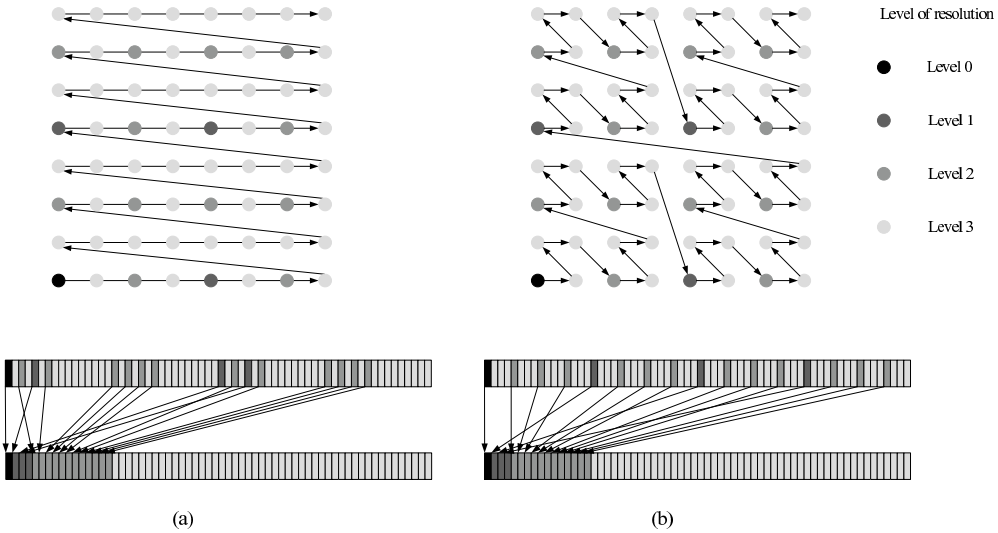


Figure 6.2: The relation between the one-dimensional index and the hierarchical index, using (a) scan line order or (b) Z-order [107].

Once the hierarchical index has been computed for each data point, the data set can be reordered and stored accordingly. The Z-order index is only used intermediately to facilitate the computation of the hierarchical index. The final translation from (x, y, z)-coordinates to the hierarchical index can be stored in a lookup table.

6.3 Design decisions

We have implemented this method and extended it to time-dependent data sets [107]. There were, however, several design issues that had to be resolved before doing this:

- Time vs. Space: how to incorporate the temporal dimension?
- Interactivity: automatic resolution switching.
- Defining a region-of-interest.
- Managing a time window.
- Using multiple threads.

In the following Subsections, each of these issues will be addressed briefly.

6.3.1 Time vs. Space

The Z-order indexing and hierarchical indexing algorithms can be easily extended to four dimensions. Examples were shown for two and three dimensions (Figures 6.1 and 6.2), but the algorithms work just as well for n-dimensional spaces.

However, suppose time is treated in the same way as space, that is, a time-dependent data set is treated as a regular 4D data set. In Section 3.5, some advantages and disadvantages of such an approach were already discussed in the comparison of several data structures. Here we will further elaborate on those considerations.

In the current algorithm all dimensions will be handled equally. This means the resolution of the data will be the same at all time in all (four) dimensions. As a consequence, if the spatial resolution of the data set is lowered, the temporal resolution is lowered as well; and if all time steps are to be shown, the full spatial resolution has to be used as well. Furthermore, not only the resolution of the data set has to be the same, also the size of the data set has to be equal in all dimensions. This may be even more of a problem. Data sets resulting from CFD simulations can have a spatial size of 128^3, for example, but consist of $5,000$ time steps. Such a data set cannot (easily) be represented as a 4D multi-resolution data set by the current algorithm.

The other extreme is to treat time and space differently and use the hierarchical indexing method only for the three spatial dimensions. Each individual time step will be processed as a separate three-dimensional data set and for each of these time steps, a single multi-resolution representation will be created. No use will be made of temporal coherence or of compression in the time dimension. Of course, with respect to the size of the dimensions, the same restriction as mentioned above still holds: the size of the data set has to be the same in all (three) spatial dimensions. In the discussion in Section 6.5, a workaround for this restriction is presented.

This approach, of treating the data set as a number of individual time steps and creating a separate multi-resolution representation of each of these time steps, is the most straightforward extension of the original algorithm to time-dependent data sets. It is the approach we have selected for implementation, the main reason being the drawbacks of the other approach mentioned above; we feel the user should be able to change the spatial and temporal resolutions separately.

Although it would be *elegant* to have all dimensions being treated equally, that is not the way in which we normally look at the data. In general, a time-dependent data set will be viewed as a movie — whether or not an interactive one. The data is viewed one time step at a time, but the three spatial dimensions are viewed simultaneously. Stated otherwise, in four-dimensional space, slicing is performed, but always perpendicularly to the time axis.

Of course, there will always be exceptions for specific types of applications. You can

think of a 3D image, where one of the dimensions is time. This would represent the temporal evolution of an arbitrary two-dimensional slice of the data set, and could be created by performing slicing in 4D, parallel to the time axis. Another example where a data set is not accessed one time step at a time is with four-dimensional isosurfacing [101], e.g. for the purpose of feature tracking. See also Section 2.5.1.

However, in general, most applications will treat a four-dimensional data set as a series of time steps. Therefore, the approach of applying the hierarchical indexing algorithm to every single time step separately is justified. Three-dimensional space is stored in a multi-resolution data structure, but time is not. Changing the spatial resolution can be easily done using this algorithm. Changing the temporal resolution can then be done by subsampling in time, i.e. skipping individual time steps.

6.3.2 Automatic resolution switching

We developed a system for multi-resolution visualisation of large time-dependent data sets with functionality for space-time navigation. One of the main goals was to have interactive visualisation. "Interactive" in this case will be defined as a user-provided desired frame rate. This quickly leads to automatic switching between resolutions. The system will automatically adapt the resolution of the data in order to achieve this frame rate. If the desired frame rate cannot be sustained for a certain amount of time or for a number of frames, the resolution will be lowered. This process will repeat until the desired frame rate is achieved. In practice, if the resolution or the desired frame rate is too high, initially, the frame rate will gradually increase while the resolution is lowered automatically, until the frame rate reaches the desired value.

On the other hand, if the desired frame rate can be achieved with a large enough margin, the resolution will be raised. (See Figure 6.3 on the facing page.) This is somewhat more complicated, because it is difficult to specify what a large enough margin is. For that, it should be possible to predict what the frame rate will be at a higher resolution. Theoretically, the frame rate at a certain resolution will be about one eighth of that at the half resolution, but this may well depend on the application, the data set, or other factors. It is important that great care should be taken to prevent undesirable switching back and forth between resolutions. Therefore, the margin for switching to a higher resolution should be taken large enough.

Naturally, it should always be possible for the user to enforce a certain high resolution, independent of the attained frame rate. However, the ability to specify a minimal frame rate is a very useful and highly desirable function, especially in an interactive application.

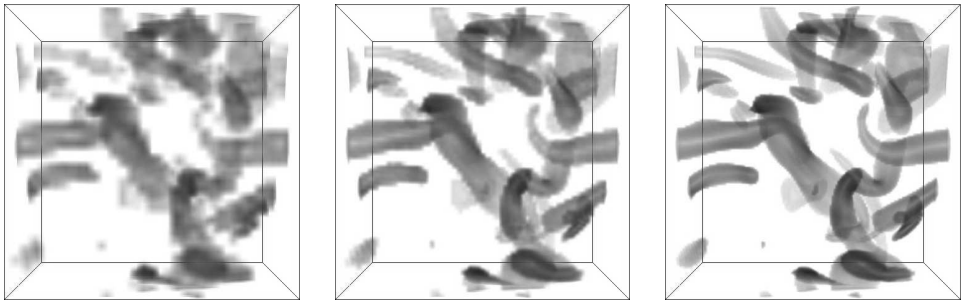


Figure 6.3: A volume rendering of the vortices data set at increasing resolution from left to right. See also colour Figure C.11.

6.3.3 Region of interest

Another addition is the possibility of defining a region of interest, to reduce the amount of data that has to be visualised. By selecting only part of the spatial domain, the data that is of interest can be visualised at a higher resolution, with the surrounding data, if necessary, as a low resolution context. See Figure 6.4.

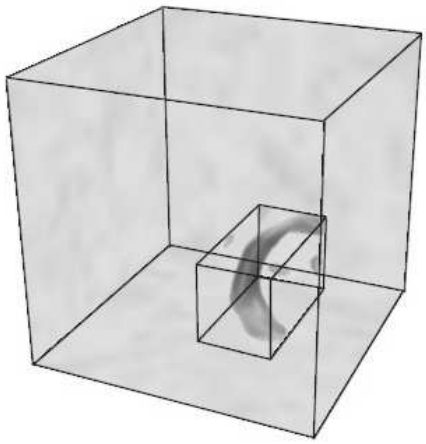


Figure 6.4: The selected region of interest is visualised using high resolution. The surrounding data is also shown for reference, but in lower resolution. See also colour Figure C.12.

In our application, an axis-aligned box is used for selecting the region of interest. This method was chosen for the ease of implementation, but naturally other methods could also be added. For the data resolution inside the region of interest, the same

principles are used for automatic resolution switching that were discussed in the previous Section. The data outside the region of interest are shown in lower resolution, or can be left out entirely.

Obviously, the motive for defining a region of interest is data reduction. Although this data reduction can speed up the rendering, the hierarchical indexing algorithm was not designed for region-of-interest selection. Refer to Figure 6.2b. If the *entire* data set is visualised, at *any* resolution, the data points needed can be read consecutively from the hierarchical data structure. However, an arbitrary region of interest will in general consist of multiple non-consecutive blocks (or even single values) on disk. Therefore, region-of-interest selection will partly undermine the advantages of the hierarchically indexed data structure.

There are two ways to do this selection, which we have both implemented. First, indexing of the data points within the region of interest can be done in main memory. This means that the entire data set has to be read from disk, before the selection can be done. After the selection, the data outside the region of interest can be discarded. The second method is to do the indexing in external memory. In this case, only the data points within the region of interest will be read from disk and therefore, much less data has to be transferred to main memory. However, because the required data points will be scattered throughout the data set, the transfer will be very inefficient. A comparison of the speed of both methods is presented in Section 6.4.

6.3.4 Time window

To further optimise the visualisation, a time window approach is used, similar to the one described in Chapter 5. A time window is basically used for buffering of time steps. Rendering is done on one time step within the time window, while in the background other time steps are read from disk to fill up the time window buffer. (See also Section 6.3.5.)

Ideally, the time window size would be specified as an amount of memory. Depending on the amount of used or available memory, new time steps could be read or old time steps could be discarded. However, managing such a time window is not straightforward.

A time window size that is specified as a number of time steps is much easier to implement, but the total amount of memory used may vary substantially, especially when switching between resolutions. The time window could be specified by a first, last and current time step, for example. This depends somewhat on the use of the data set. If playing in reverse time direction is not desired, there is no need to keep a history of the time steps already visited. But if reverse playing is as common as forward playing, it might be more useful to have a time window that is centred on the current time step.

In our application, the time window size is specified as a number of time steps. The structure of the time window is a circular doubly-linked list. Two pointers into this list keep track of the current time step that is being rendered and the last time step that has been read. These pointers will shift along the linked list in the same direction. However, they will move independently, depending on the rendering speed and the reading speed, respectively.

Because most of the time these two pointers will point to different time steps, there is a delay between an action being taken by the user (for example, increasing the resolution) and the result of this action being shown to the user. Suppose the time window is filled with time steps that have been read at a certain resolution. If the resolution is changed by the user, the time steps that have already been read into the time window will simply be visualised at the resolution they were read at. The newly read time steps will be read into the time window at the new resolution and visualised at that resolution whenever they become the current rendering time step. The delay will of course depend on the difference between the two pointers and on the frame rate.

Also when the resolution is raised automatically by the program, the same delay will occur. Higher-resolution data will be read from disk, but only after some time will the user see the result, because the reading time step is ahead of the rendering time step.

Only when the desired frame rate cannot be achieved, and therefore the resolution is automatically lowered by the program, then the program may lower the resolution of the current rendering time step. First, however, the resolution of the reading time step will be lowered. Then, if the frame rate is still too low, the rendering resolution of the current time step will be lowered. Remember, this time step was stored in the time window at a higher resolution, but because the desired frame rate cannot be achieved, the rendering resolution will be lowered. After that, if necessary, the reading resolution will again be lowered, and so on.

There are more considerations to be made in the design of a time window, such as regarding the region of interest, playing backward and forward in time, or changing the temporal resolution. The specific implementation may vary depending on the application or the desired use.

If enough memory should be available, a time window could be designed to keep multiple levels of resolution for every time step. Such a time window would resemble a 5D scale-space [4]. This would mean the time window could always keep the lower levels of resolution available, so that interactive playing can be quickly resumed at any time. Having a time window that spans several time steps and several resolutions will make truly interactive space-time navigation possible.

6.3.5 Multi-threading

While the user is interacting with the data, the program runs in the background reading data from disk in order to fill the time window. To do this, without the interaction being disturbed too much, a multi-threaded approach is applied. Two threads are running simultaneously, one to fill the time window with new data read from disk, and another for visualising the data from the time window.

Idle time is used to read ahead, to make the playback more smooth. Idle time can also be used to read higher-resolution data for the time steps within the time window. In pause mode, while the user is examining the data of the current time step, the resolution of the data can be raised in the background. As soon as the higher resolution data is available, the visualisation thread is signalled and the data is visualised. The user will see the image gradually improving with each next level of resolution.

6.4 Results

In our application, two threads were used, as described above. The time window can be specified as a number of time steps, and a region of interest can be selected if needed, by defining two diagonally opposite corner points of an axis-aligned box. For rendering we have implemented two methods. The first one uses three axis-aligned orthogonal slice planes, the second one is a simple volume rendering technique using 2D texture-mapped semi-transparent slices.

We have made no use of temporal coherence or any other type of compression. Every time step of the time-dependent data sets has been individually reorganised using the hierarchical indexing algorithm described in this Chapter.

We have used two data sets for benchmarking, both of a fluid dynamics simulation containing turbulent vortex structures. The first data set has a resolution of 128^3 and consists of 100 time steps. The total size for this data set is 800 MiB. The second is a supersampled version of this same data set, with a resolution of 256^3 and 200 time steps, totalling 12.5 GiB.

The benchmarks were performed on a Dual Athlon MP 1.2 GHz machine with 3 Gigabytes of main memory and a GeForce 3 graphics card.

Data set	Conversion (s)	LUT generation (s)
128^3	77	0.50
256^3	1620	4.39

Table 6.1: The time needed to convert the original data set to the multi-resolution representation and to generate the lookup table.

First, the conversion time from the original data set to the hierarchical multi-resolution representation is shown in Table 6.1 on the facing page. Naturally, this time heavily depends on the I/O transfer rate. This conversion can be done in preprocessing and consists of reading the original data from disk, rearranging the data in memory, according to the hierarchical index, and writing the reordered data to a new disk file.

Also shown in this table is the time needed to generate the lookup table, introduced in Section 6.2. This lookup table will speed up the conversion in run time from the (x, y, z)-coordinates to the hierarchical index. The size of the lookup table will be typically in the order of the size of one time step. Although the computation of the hierarchical index can be performed very quickly, for a time-dependent data set these computations are the same for every time step. The construction of this lookup table can be done in preprocessing and it will speed up the conversion of the coordinates in run time. On the other hand, for very large size data sets, the lookup table may have to be stored in external memory. For example, a 1024^3 data set will result in a lookup table of 4 GiB in size. In that case, computation of the hierarchical index on-the-fly will be much faster than reading from a lookup table stored in external memory. From Table 6.1 on the preceding page it can be computed that the conversion time for a single data point is about 0.25 μs.

Data set size	Slice rendering (fps)	Volume rendering (fps)
16^3	123	42
32^3	106	18
64^3	33	4.0
128^3	3.3	0.6
256^3	0.2	0.07

Table 6.2: Average frame rates achieved using different spatial resolutions, both for slice rendering and volume rendering.

Table 6.2 shows the average frame rates that can be attained while playing back the data set at several resolutions, both using slice rendering and volume rendering. These measures are done without the use of a time window, and without specifying a minimal frame rate.

The rates are found by determining the minimum of the rendering frame rate and the reading frame rate and averaging these values over time. In practice, the playing frame rates can turn out to be higher through the use of the time window. Because a number of time steps will have been buffered in the time window, the rendering will not be slowed down by the reading. For example, with a time window of size 50, and the minimal frame rate set to 5 fps, the 128^3 data set can be played back at full resolution, using slice rendering, although Table 6.2 indicates a frame rate of 3.3 fps.

Table 6.3 on the following page shows, for various resolutions, the time in one reading iteration and the time needed to rebuild a regular data structure from the hierarchi-

Data set size	Reading iteration time (ms)	Rebuilding time (ms)
16^3	1.8	0.2
32^3	11.6	2.1
64^3	41.4	16.0
128^3	289.4	168.5
256^3	2081.8	1348.9

Table 6.3: The reading time and rebuilding time in milliseconds for various resolutions.

Resolution	Data set size (MB)	Memory usage (MB)
16^3	0.156	18
32^3	1.25	20
64^3	10	31
128^3	80	117
256^3	640	797

Table 6.4: The memory usage of the application for different resolutions, using a time window of 10 time steps.

cally indexed data. Especially for large resolutions, the rebuilding time constitutes a substantial percentage of the total reading time.

This observation leads us to devise an extension to the standard hierarchical indexing algorithm, that may speed up visualisation of the full resolution data set. The change would be to store all but the highest resolutions using the hierarchical index, and store only the highest resolution in "normal" scan line order. This would mean a space overhead of 12.5%, but it would remove the time-consuming reconstruction of the full data set from the hierarchical index.

The memory usage of the application has also been measured. For different resolutions of the data set, the amount of memory used by the application was determined, using a fixed time window of size 10. Table 6.4 shows, for each resolution, the amount of memory occupied by the 10 time steps in the time window, and the total memory usage of the program. Besides the fixed overhead in memory, mainly for the user interface of the program, there is a variable (data size related) overhead consisting of the lookup table, the data used by the renderer, et cetera.

The region-of-interest performance was also tested. A fixed size region of interest was defined (32^3) in different size data sets. As was explained above, the hierarchical data structure has not been designed for region-of-interest selection. The data points from within the region of interest have to be selected from the highest resolution hierarchical data set. This indexing can be done on disk, which is slow, but saves unnecessary data transfer, or in memory, which is a lot faster, but requires the entire highest resolution data set to be transfered to memory first. We have tested both

Data set size	Indexing in memory (fps)	Indexing on disk (fps)
64^3	84.4	10.4
128^3	17.8	10.8
256^3	0.4	10.0

Table 6.5: The region-of-interest rendering frame rates, for a fixed (32^3) region of interest and various data set sizes. The performance is tested both with indexing in memory and indexing on disk.

methods; the results are shown in Table 6.5.

With indexing in memory, the frame rate drops as data set size increases. The entire high resolution data set has to be read from disk, therefore, with large data set sizes, disk access will become a bottleneck. With indexing on disk, the amount of data that has to be transfered only depends on the size of the region of interest. Therefore, the results are similar for different size data sets. For small data set sizes, the indexing on disk will become a bottleneck, because it would be quicker to simply load the entire data set into main memory. However, for large size data sets, indexing on disk will be faster than indexing in memory.

6.5 Discussion, limitations and extensions

Because of the design of the Z-order space-filling curve, the Z-order indexing algorithm is in principle limited to data sets that have equal dimensions that are a power of 2, e.g. 64^3 or 256^3. If a data set has other dimensions, two straightforward solutions are to pad the data with zeros (or similar) and/or to split the data set in smaller parts.

Padding with zeros (or some sort of coding value, such as NaN) up to a correct (2^n) size is easy, but might generate a lot of space overhead. For example, a $90 \times 60 \times 60$ data set would have to be padded up to $128 \times 128 \times 128$, which is more than 6 times larger. Padding could be combined with splitting the data set in smaller parts. A $256 \times 128 \times 128$ data set can easily be split into two 128^3 data sets. The $90 \times 60 \times 60$ data set can then be padded up to $96 \times 64 \times 64$ and split into one 64^3 and four 32^3 data sets. Figure 6.5 on the next page shows an example in two dimensions.

Although this is a very strict limitation of the data set size for the Z-order indexing algorithm, it doesn't have to be the case for the hierarchical indexing algorithm. When a lookup table is used for the hierarchical index, the dimensions of the data set do not matter. So, perhaps it is necessary to pad the data set during preprocessing, to compute the Z-order index. When the hierarchical index is computed, the padding regions can be skipped. A lookup table is generated to transform the spatial coordinates into a hierarchical index. No more data size restrictions apply. Only some

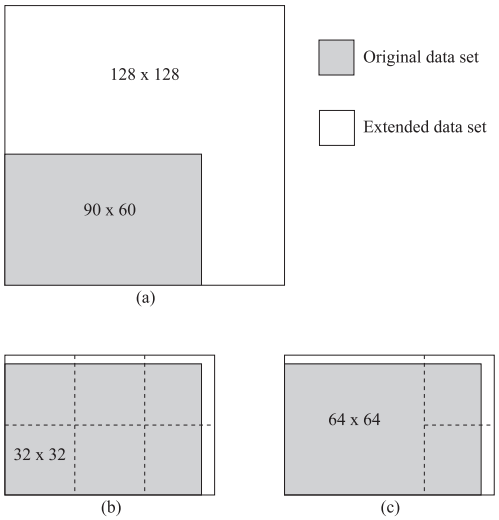


Figure 6.5: A two-dimensional 90×60 data set has to be (a) padded up to 128×128, (b) padded up to 96×64 and split into 6 times 32^2, or (c) padded up to 96×64 and split into one 64^2 and two 32^2 parts [107].

extra administration has to be kept for the number of data points at each level of resolution, as this will no longer be straightforward.

6.5.1 Downsampling vs. subsampling

Lowering the resolution of a data set can be done in several ways. From sampling theory we learn that a low-pass filter should be applied before downsampling. The low-pass filter functions as an anti-aliasing filter to reduce the bandwidth in order to satisfy the Nyquist-Shannon sampling theorem [78]. The simplest form of a low-pass filter is just the averaging filter.

We are not interested in arbitrary downsampling. All we want to do in this application is to half the resolution. Therefore, by averaging every two data points, a downsampled, anti-aliased version of the data set is constructed. If we don't care about the sampling theorem, an even more simple method of downsampling is subsampling: to take every second (or nth) data point.

This is the way in which the hierarchical indexing algorithm works. The low resolution versions of the data are found by taking every nth data point of the original data. Every second data point is used for the half resolution data set, every fourth data point is used for the quarter resolution data set, et cetera.

The advantage is that no data is replicated. The multi-resolution data set is exactly the same size as the original data set. Furthermore, the low resolution data can be reused in a higher resolution version. This means that less data has to be read when switching to a higher resolution.

It is possible to do downsampling by averaging, and store the averages as the low resolution data. However, the original high resolution data then has to be stored as well, increasing the size of the multi-resolution data set as a whole, or, alternatively, the original data has to be reconstructed using the averages. This will cost some processing time. However, the resulting visualisation will be of better quality, because the low resolution data is smoothed. Such a technique could be compared to mipmapping [105].

The same goes for downsampling in time, as was discussed above. Currently we use subsampling in time, which means that every nth time step is visualised. Here also averaging could be used instead of subsampling, to get nicer results. By storing the averages for every two time steps, the temporal resolution could be halved. However, this would again involve an increase in the data set size, and it would introduce a new artefact: motion blur. These averaged time steps should only be used if the desired frame rate cannot be achieved. By using the halved temporal resolution, the playing frame rate is effectively doubled. However, should the playback be paused by the user, one of the original time steps should be visualised instead of the time-averaged one, because of the motion blur.

The most straightforward way to construct a complete multi-resolution data set in both space and time, using averaging, would be to precompute and store the downsampled data at every possible spatial and temporal resolution. The disk space needed for storing all spatial resolutions would be:

$$1 + \frac{1}{(2^3)} + \frac{1}{(2^3)^2} + \frac{1}{(2^3)^3} + \frac{1}{(2^3)^4} + \cdots \approx \frac{8}{7} \tag{6.3}$$

which is about 114% of the size of the original data, and an overhead of approximately $\frac{1}{7}$. The disk space needed to store all temporal resolutions would be:

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots \approx 2 \tag{6.4}$$

times the size of the original data set. Multiplying these two numbers, we get a total size of the multi-resolution data set of approximately 228.5% of the original data set, or an overhead of 128.5%.

6.5.2 Compression

Another possible extension to the normal hierarchical indexing algorithm could be to make use of temporal coherence. Instead of storing every time step completely, using

some sort of compression could reduce the data set size substantially. A possible solutions would be to store differences between time steps, which can be encoded using fewer bits than the original data values.

Video compression techniques such as MPEG [46, 47] can be a source of inspiration for temporal compression algorithms. The concepts of both forward and bidirectional predictive coded frames could perhaps be employed in the context of 3D time-dependent data sets.

Compression in the spatial domain seems to be difficult to combine with this multi-resolution data structure; points that are located close to each other in the spatial domain, may be scattered throughout the multi-resolution representation.

CHAPTER 7

Conclusions and future work

7.1 Conclusions

The research in this project was directed towards visualisation of very large, time-dependent data sets. In particular, my focus has been on interactive visualisation of surfaces within these data sets. Obviously, large data handling and interactivity are in general not easily combined.

Several approaches to handling and visualisation of time-dependent data sets have been explored in this thesis.

Feature-based visualisation (see Chapter 2) is a very useful approach. By selecting the interesting parts of the data beforehand, a huge data reduction can be achieved. Furthermore, the selected data can be further reduced by abstracting the features that are of interest. The remaining data can then be handled efficiently and interactively. The features can be explored and their evolution over time can be studied. Moreover, by abstracting the data, the features of interest are described quantitatively, helping greatly in studying the evolution of the features. On the downside, by abstracting the data to features, a lot of information is lost. The criteria for selecting the features have to be very good, because the abstraction cannot be undone: from the features, the original data cannot be retrieved. If other features have to be extracted, or the selection criteria have to be changed, the original data has to be recovered and

the whole selection-abstraction process has to be repeated. Abstraction of the data also limits the possibilities for visualisation and querying. All the data that might be of interest during visualisation has to be selected and stored with the feature data. Finding the right criteria for the features of interest naturally requires extensive knowledge of the application in question, but it might still take some time fine-tuning them. The drawback of this is that the extraction process has to take place on the original data; where this requires interaction with the user, the large data handling problem still remains. However, once the data has been reduced and the features have been extracted, very useful and interesting visualisations can be made and interactive exploration of the data can be performed.

In the context of this research project, although interesting at first, we did not explore this approach any further. Because of the developments within the project by our co-researchers (see Section 3.1), the use of feature-based techniques was no longer necessary.

The second approach that was discussed in detail is the use of special, fast-access data structures. Such data structures are designed to perform a certain visualisation task efficiently. The data structures make use of the properties of a particular visualisation algorithm and are made to fit the algorithm exactly. Data structures exist for fast volume rendering or for fast isosurface cell search, for example. These data structures are created in preprocessing. On using these data structures, the performance will usually be much better than that of the standard visualisation algorithms.

One particular data structure was discussed in detail in Chapters 4 and 5: the temporal index tree, in combination with the interval tree. The interval tree was designed for very fast isosurface cell selection. This data structure is combined with a temporal index tree that makes use of data coherence in time to compress the data set in this dimension. The combined application of these data structures results in high speed isosurface cell extraction from time-dependent data sets. We have developed an incremental cell search algorithm to speed up the extraction even more in browsing through time. The extraction algorithm was combined with a special-purpose fast direct-rendering algorithm in order to prevent the rendering from becoming the bottleneck.

We have succeeded in lowering the huge memory requirements by creating an out-of-core version of the index tree. Through the use of a time window, keeping only part of the tree in memory, we have enabled interactive isosurfacing on a normal PC.

The advantage of data structures like these is that they are designed to perform a certain visualisation algorithm very fast. However, the drawback is that they are also usually limited to perform only that algorithm.

For extracting isosurfaces from time-dependent data sets, the index tree works very well, but the resulting visualisation cannot be combined with another type of visual-

isation. The data is stored in such a way in the index tree, that only isosurface cells can be extracted from it. The original, raw data set is not available, so the isosurface cannot be coloured with another scalar, or combined with streamlines, for example. Only through (time-consuming) approximate reconstruction of the data it is possible to use the scalar field for another visualisation such as slicing, as was shown in Figures 5.1 and 5.5.

Creating such a data structure will take some time in preprocessing, but if you need one specific type of visualisation, that will certainly be worth the waiting, because the performance of the data structure will be very high. However, if a combination of visualisation techniques is needed, the applicability of such data structures will probably be too limited.

Some minor extensions to this data structure may be possible. It would be interesting to research the possibility of further compressing the data structure, for example, in a way similar to the technique presented by Bordoloi and Shen [5]. To the standard data structure they added compression using transform coding, a technique originating from the field of image processing.

Extending the index tree in other ways, such as adding spatial compression, adding data for volume rendering or particle tracing, or creating a multi-resolution version of the index tree, I believe, is hardly possible.

The final approach that was discussed is the use of multi-resolution data structures. These structures enable the data to be accessed at several levels of resolution. The original data can be retrieved, but if available time or memory is limited, a lower resolution version of the data can also be requested. The lower resolution data can be a downsampled or subsampled version of the original data. Whatever resolution is used, raw data is returned, which means that the visualisation is as flexible as with the original data. Unlike the fast-access data structures described above, (most) multi-resolution structures are not designed for one specific visualisation algorithm. This is a large difference between the two approaches, and a huge advantage of the latter. Naturally, for particular visualisation tasks, fast-access data structures will often be faster than multi-resolution structures. However, a multi-resolution data structure provides the flexibility to switch between different visualisations and is designed to handle large data sets by trading off data resolution for speed.

There are many different ways to create a multi-resolution data structure. The method that was discussed in Chapter 6 reorders the data in such a way that low resolution (unfiltered, subsampled) data is at the front of the data set and data points from consecutive higher levels of resolution are stored consecutively on disk. This has a number of advantages. First, there is no data replication, so the size of the multi-resolution data set is the same as that of the original data set; there is no spatial overhead. Secondly, data points from a single level of resolution are stored coherently on disk and therefore cache-friendly. Thirdly, because the data is merely reordered,

lower resolution data is part of the higher resolution data. This means that the lower resolution data can be reused and less data has to be read when increasing the resolution. The fact that the low resolution data is only a subset of the high resolution data is also a disadvantage: if the data would be downsampled (averaged) instead of subsampled, less information would be lost in the lower levels of resolution, resulting in the data and visualisation being more smooth.

We have extended the existing multi-resolution data structure to time-dependent data sets. We have presented this data structure in combination with techniques for region-of-interest selection, time window management and automatic resolution switching providing interactive visualisation and space-time navigation of these large 4D data sets.

7.2 Directions for future work

Each of the approaches discussed has its advantages and disadvantages and each has its specific application areas. Therefore, it is difficult to specify one technique, or even one approach that will solve all our problems.

There will always be a trade-off between applicability and performance. Depending on the requirements of the application, one of several approaches will have to be selected. The first approach would be that of fast-access data structures. However, as I already mentioned in the previous Section, I believe the index tree data structure cannot be extended much further. Therefore, my recommendations for further research will be directed towards the following approaches.

The first approach is the use of generic, multi-resolution data structures. The obvious advantage over specific, fast-access data structures is the greater applicability of such a data structure. On the other hand, data resolution will be traded off for speed, if necessary. The multi-resolution hierarchical indexing data structure from Chapter 6 is the first of my recommendations. This is an interesting technique that provides multi-resolution data access without any spatial overhead. However, it does this by subsampling (without filtering) instead of downsampling or averaging the data. (See also Subsection 6.5.1.) This means that the data at low resolution is also of low quality. A worthwhile extension of the algorithm would be to improve the quality of the data at lower resolutions by filtering the data before downsampling. By storing averaged data instead of subsamples, the resulting low resolution visualisations will be much better and more useful. The averaged data can be stored instead of, or together with the original, lower resolution data. It will either cost more processing time to reconstruct the original data from the averages, or it will cost more disk space to store both the original and the averaged data. In both cases, the advantages of the original algorithm will be partly undone.

Adding compression to this data structure is also worth investigating. Neither spatial nor temporal compression is used in the standard algorithm. Adding spatial compression might be difficult because of its design concept of reordering the data. On the other hand, if averaged low resolution data is used, as suggested above, it may prove to be effective to compress this data, for example using difference encoding. Furthermore, the extension to time-dependent data could be easily combined with temporal compression. A straightforward approach would be the use of delta encoding in the time dimension, as well.

More in general, inspiration for temporal compression can be sought in video compression algorithms such as the well-known MPEG coding systems [46, 47]. Interesting concepts found in video compression include motion compensation and both forward and bidirectionally predictive-coded frames.

The Wavelet-based Time-Space Partitioning or WTSP Tree [100], which was briefly introduced in Chapter 3, is another example of a data structure from the same approach. But not only does it provide multi-resolution access in both the spatial and the temporal domain, it also makes use of compression techniques. Wavelet transforms are used both in space and in time and, on the resulting coefficients, run-length and Huffman encoding is applied. This is a great example of the combined use of several techniques from different approaches. However, the WTSP Tree was designed for volume rendering. It is not a generic data structure, suitable for any type of visualisation algorithm. The data that is stored in the tree nodes does not allow for isosurface extraction or particle tracing, for example. Extending the WTSP Tree in this manner, enabling other types of visualisation algorithms, is a most promising direction for future research.

Wavelet-based techniques, in general, are good candidates for visualisation of large data sets, because on the one hand the wavelets can be used for compression of the data, and on the other hand, they provide multi-resolution access to the data.

The second approach is feature-based visualisation. This too is an exciting and very promising research field. A lot of interesting work has already been done, the results of which are impressive. The approach is more or less separate from all other approaches, because it handles features, not data. There is a layer of abstraction over the raw data, therefore it is difficult to combine this approach with many of the others.

In feature-based visualisation, there is always the need for the detection of new types of features, such as boundary layers, recirculation zones and phase fronts, and the development of new selection and extraction criteria. Also, techniques for tracking of these features and for the detection of new types of events have to be developed.

Feature extraction can be combined with multi-resolution and multi-scale techniques. Features extracted in lower resolution data can be used to aid the extraction in higher resolution data. Also, when noise is present in the data, it can be essential to perform

filtering before the extraction, in order to find meaningful features. The combination of feature-based visualisation with multi-scale techniques, finally, is very interesting, enabling tracking of features not only in time but also in scale-space, as presented by Bauer and Peikert [4].

The bottleneck in feature-based visualisation is in the first steps of feature extraction. Because the feature extraction is performed on the original data, the large data handling problem remains. Interactive feature selection or extraction techniques have to be developed, such as the Focus+Context technique by Doleisch et al. [12]. In some cases the feature extraction can be performed without any user interaction and therefore fully automatically. If not, it should be possible to do the extraction interactively, or at least the selection, or the development of the extraction criteria, should be possible interactively. In the latter case, the actual extraction process can be performed offline.

Having extracted the features from a data set, the feature visualisation could be used as a guideline for browsing the data in some other form. Further research should be done into the combination of feature visualisation and other techniques, such as isosurfacing and volume rendering, and their interaction. By focusing on the features, or selecting data guided by the features, a substantial reduction of the raw data can be achieved. This reduced amount of data, containing the features of interest, can then be used for further investigation. The features can be visualised as icons, slicing or volume rendering can be used to show the raw, selected data within and around the features, streamlines can be seeded from the surface of the features, et cetera. As an example of an existing technique, making use of such combined visualisations, see the vortex browser by Stegmaier et al. [86].

Alternatively, the features may be used as an index, for example, for an adaptive compression algorithm. Features will indicate the location of the interesting data, influencing the amount of compression for that data. Similarly, significant events in the temporal evolution of the features can determine which time steps to use for key frames.

Combining different techniques and algorithms, even from different approaches, may not always be possible, and often far from straightforward, but most certainly is highly desirable, if not essential, for interactive visualisation of the data sets of the future.

Colour Figures

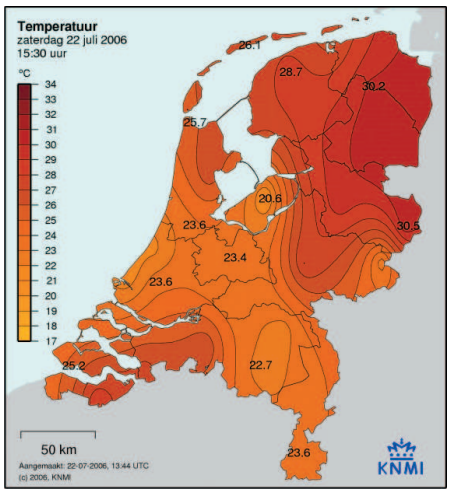


Figure C.1: A temperature table and accompanying temperature map for The Netherlands. (Source: KNMI)

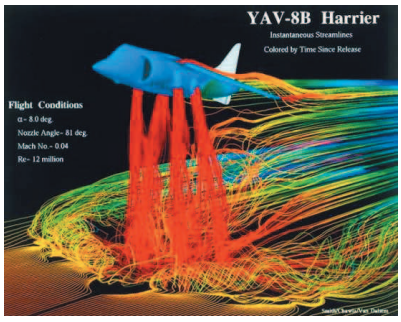


Figure C.2: An example of flow visualisation. The flow around a Harrier aircraft, shown using streamlines. The colour indicates the time since "release". (Source: aerospaceweb.org)

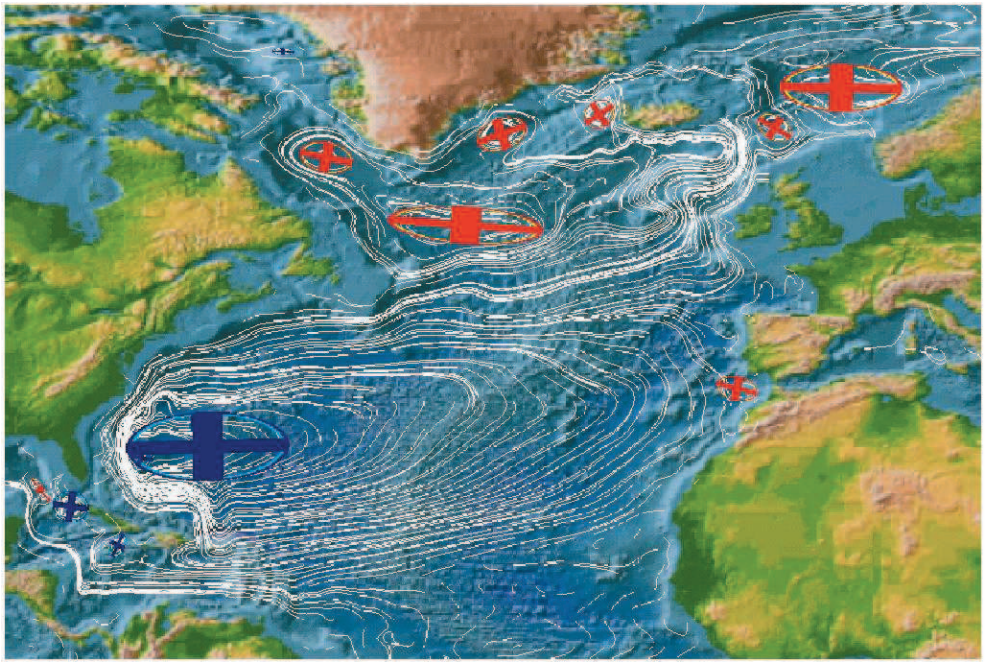


Figure C.3: Flow in the Atlantic Ocean, with streamlines and ellipses indicating vortices. Blue and red ellipses indicate vortices rotating clockwise and counterclockwise, respectively [72].

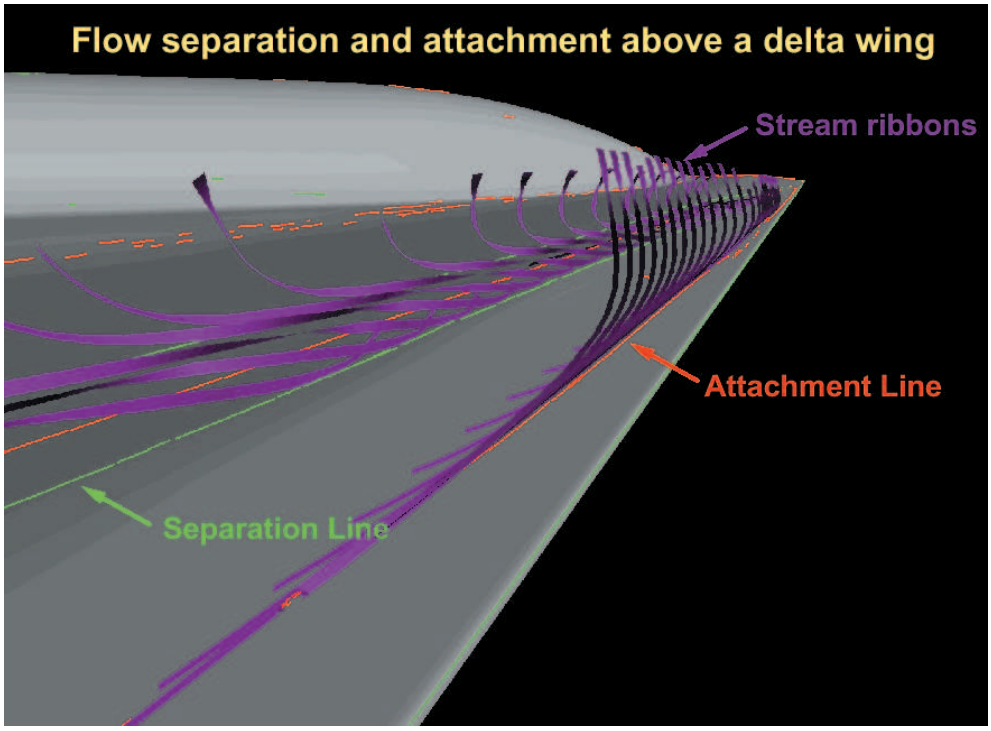


Figure C.4: Separation and attachment lines on a delta wing (Source: D. Kenwright).

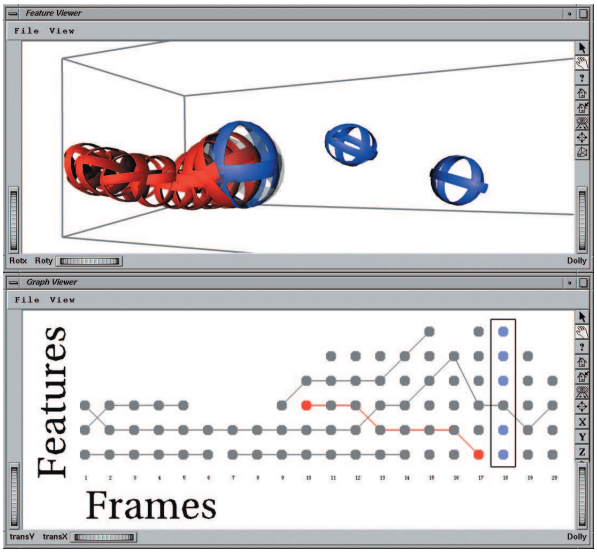


Figure C.5: One step during feature tracking. A path is shown with its prediction, and three candidates in the next time step [60].

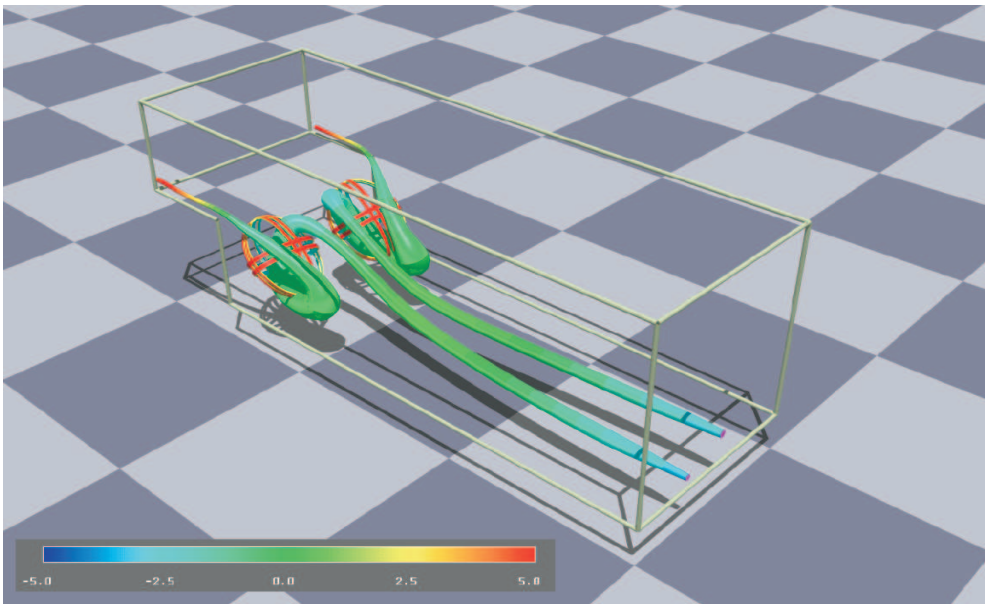


Figure C.6: Visualisation with streamtubes of the recirculation in the backward-facing-step data set [99].

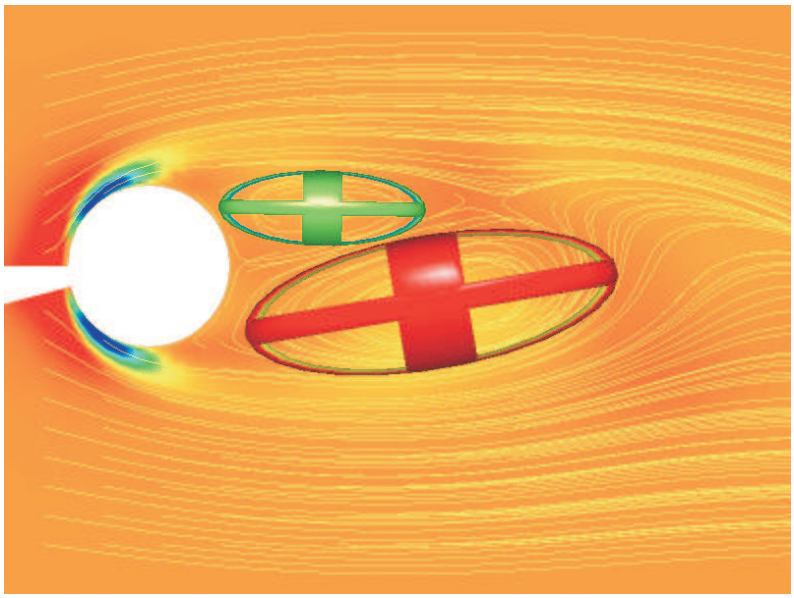


Figure C.7: Vortices behind a tapered cylinder. The colour of the ellipsoids represents the rotational direction [71].

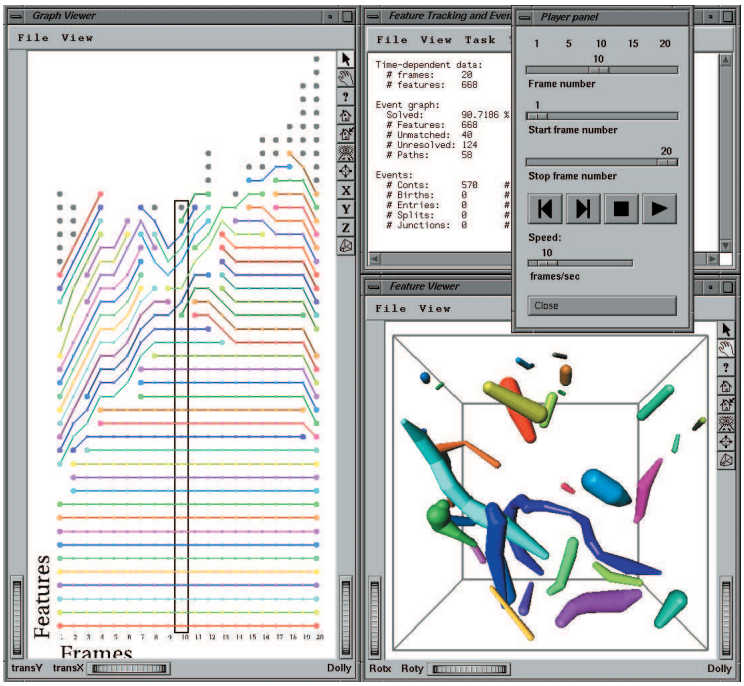


Figure C.8: Playing through the turbulent vortex data set.

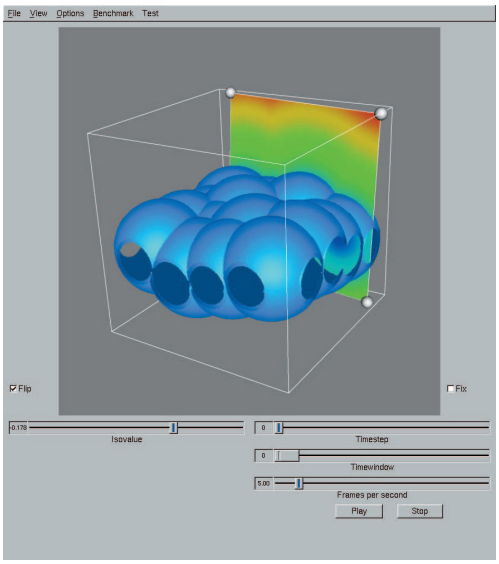


Figure C.9: A 256^3 data set of air bubbles rising in water.

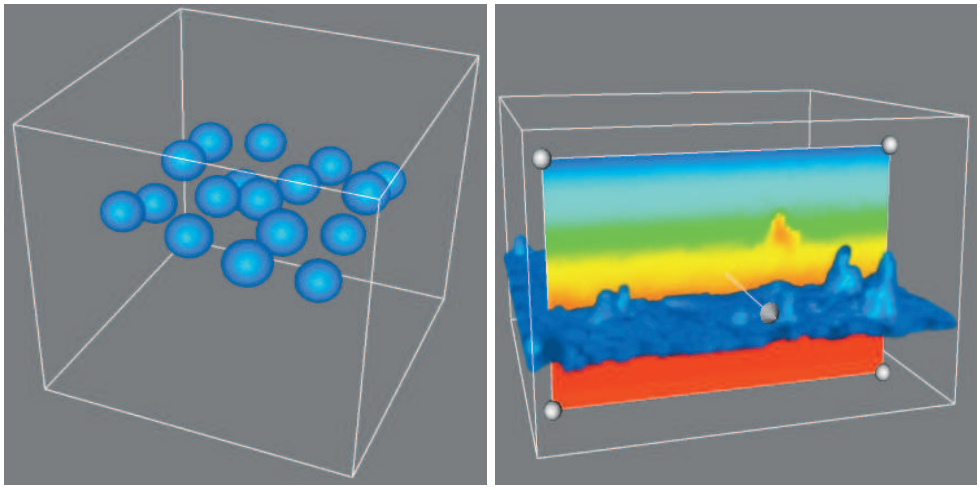


Figure C.10: Scenes from the two data sets. On the left is the bubble data set, on the right is the cloud data set.

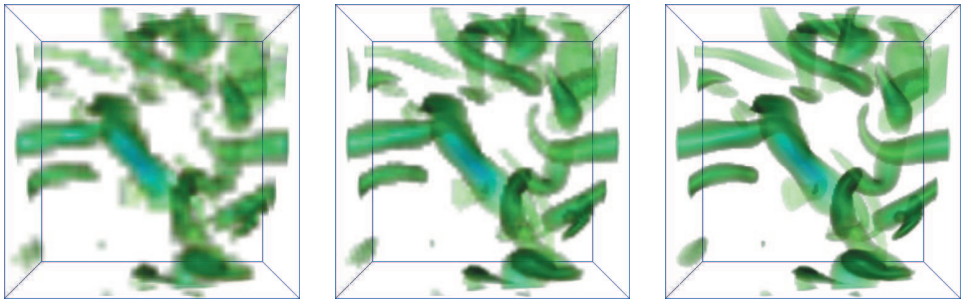


Figure C.11: A volume rendering of the vortices data set at increasing resolution from left to right.

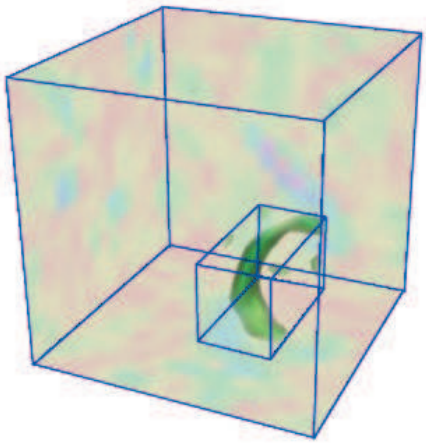


Figure C.12: The selected region of interest is visualised using high resolution. The surrounding data is also shown for reference, but in lower resolution.

Bibliography

[1] Chandrajit L. Bajaj, Valerio Pascucci, G. Rabbiolo, and Daniel R. Schikore. Hypervolume visualization: a challenge in simplicity. In *Proceedings of the 1998 Symposium on Volume Visualization*, pages 95–102, Research Triangle Park, NC, USA, 19–20 October 1998. IEEE, ACM Press.

[2] Chuck Baldwin, Ghaleb Abdulla, and Terence Critchlow. Multi-resolution modeling of large scale scientific simulation data. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 40–48, New Orleans, LA, USA, 2–8 November 2003. ACM.

[3] David C. Banks and Bart A. Singer. A predictor-corrector technique for visualizing unsteady flow. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):151–163, June 1995.

[4] Dirk Bauer and Ronald Peikert. Vortex tracking in scale-space. In David S. Ebert, P. Brunet, and I. Navazo, editors, *Data Visualization 2002. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 233–240, Barcelona, Spain, 27–29 May 2002. Springer-Verlag.

[5] Udeepta D. Bordoloi and Han-Wei Shen. Space efficient fast isosurface extraction for large datasets. In *Proceedings of the 14th Conference on Visualization 2003*, pages 201–208, Seattle, WA, USA, 19–24 October 2003. IEEE Computer Society, IEEE Computer Society Press.

[6] Charl P. Botha and Frits H. Post. ShellSplatting: Interactive rendering of anisotropic volumes. In Georges-Pierre Bonneau, Stefanie Hahmann, and

Charles D. Hansen, editors, *Data Visualization 2003. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, Grenoble, France, 26–28 May 2003. ACM SIGGRAPH.

[7] Kenneth R. Castleman. *Digital Image Processing*. Prentice Hall, 1996.

[8] Yi-Jen Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proceedings of the 14th Conference on Visualization 2003*, pages 217–224, Seattle, WA, USA, 19–24 October 2003. IEEE Computer Society, IEEE Computer Society Press.

[9] Charles K. Chui. *An Introduction to Wavelets*. Wavelet analysis and its applications. Academic Press, 1992.

[10] Paolo Cignoni, P. Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April 1997.

[11] M. Dindar, A. Lemnios, M. S. Shephard, K. Jansen, and David N. Kenwright. Effect of tip vortex resolution on UH-60A rotor-blade hover performance calculations. In *54th AHS Annual Forum and Technology Display*. American Helicopter Society, Alexandria, VA, USA, 1998.

[12] Helmut Doleisch, Martin Gasser, and Helwig Hauser. Interactive feature specification for focus+context visualization of complex simulation data. In Georges-Pierre Bonneau, Stefanie Hahmann, and Charles D. Hansen, editors, *Data Visualization 2003. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 239–248, Grenoble, France, 26–28 May 2003. ACM SIGGRAPH.

[13] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.

[14] Al Globus, C. Levit, and T. Lasinski. A tool for visualizing the topology of 3D vector fields. In Gregory M. Nielson and Lawrence Rosenblum, editors, *Proceedings of the 2nd Conference on Visualization '91*, pages 33–39, San Diego, CA, USA, 22–25 October 1991. IEEE Computer Society, IEEE Computer Society Press.

[15] Benjamin Gregorski, Joshua Senecal, Mark A. Duchaineau, and Kenneth I. Joy. Adaptive extraction of time-varying isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):683–694, November 2004.

[16] M. D. in den Haak, Hans J. W. Spoelder, and F. C. A. Groen. Matching of images by using automatically selected regions of interest. In J. L. G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 27–40, Utrecht, The Netherlands, 4–5 November 1992.

[17] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, September 1910.

[18] R. B. Haber and D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In Gregory M. Nielson, B. D. Shriver, and Lawrence Rosenblum, editors, *Visualization in Scientific Computing*, pages 75–93. IEEE Computer Society Press, 1990.

[19] Robert Haimes and D. Darmofal. Visualization in computational fluid dynamics: A case study. In Gregory M. Nielson and Lawrence Rosenblum, editors, *Proceedings of the 2nd Conference on Visualization '91*, pages 392–397, San Diego, CA, USA, 22–25 October 1991. IEEE Computer Society, IEEE Computer Society Press.

[20] James L. Helman and Lambertus Hesselink. Representation and display of vector field topology in fluid flow data sets. *IEEE Computer*, 22(8):27–36, August 1989.

[21] James L. Helman and Lambertus Hesselink. Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11(3):36–46, May 1991.

[22] David A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1101, September 1952.

[23] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22(3):343–348, September 2003. Eurographics 2003 Proceedings.

[24] Martin Isenburg and Peter Lindstrom. Streaming meshes. In *Proceedings IEEE Visualization*, October 2005. To appear.

[25] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[26] Ming Jiang, Raghu Machiraju, and David Thompson. A novel approach to vortex core region detection. In David S. Ebert, P. Brunet, and I. Navazo, editors, *Data Visualization 2002. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, Barcelona, Spain, 27–29 May 2002. Springer-Verlag.

[27] Chris Johnson. Top scientific visualization research problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, July 2004.

[28] JPEG. ISO/IEC International Standard 10918-1.

[29] D. S. Kalivas and A. A. Sawchuk. A region matching motion estimation algorithm. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 54(2):275–288, September 1991.

[30] David N. Kenwright. Automatic detection of open and closed separation and attachment lines. In David S. Ebert, Holly Rushmeier, and Hans Hagen, editors, *Proceedings of the 9th Conference on Visualization '98*, pages 151–158, Research Triangle Park, NC, USA, 18–23 October 1998. IEEE Computer Society, IEEE Computer Society Press.

[31] David N. Kenwright and Robert Haimes. Automatic vortex core detection. *IEEE Computer Graphics and Applications*, 18(4):70–74, July 1998.

[32] David N. Kenwright, Chris Henze, and C. Levit. Feature extraction of separation and attachment lines. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[33] Steven Kilthau and Torsten Möller. Splatting optimizations. Technical report, Simon Fraser University, 2001.

[34] Robert S. Laramee, Helwig Hauser, Helmut Doleisch, Benjamin Vrolijk, Frits H. Post, and Daniel Weiskopf. The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2):203–221, June 2004.

[35] B. van Leer. The computation of steady solutions to the euler equations: a perspective. Technical report, University of Michigan, 1986.

[36] Willem C. de Leeuw. *Presentation and Exploration of Flow Data*. PhD thesis, Delft University of Technology, The Netherlands, January 1997.

[37] Willem C. de Leeuw, Hans-Georg Pagendarm, Frits H. Post, and Birgit Walter. Visual simulation of experimental oil-flow visualization by spot noise images from numerical flow simulation. In R. Scateni, Jarke J. van Wijk, and P. Zanarini, editors, *Proceedings of the Sixth Eurographics Workshop on Visualization in Scientific Computing*, pages 135–148, Chia, Italy, 3–5 May 1995. Springer-Verlag.

[38] Willem C. de Leeuw and Robert van Liere. Visualization of global flow structures using multiple levels of topology. In Eduard Gröller, Helwig Löffelmann, and W. Ribarsky, editors, *Data Visualization '99. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 45–52, Vienna, Austria, 26–28 May 1999. Springer-Verlag.

[39] Y. Levy, D. Degani, and A. Seginer. Graphical visualization of vortical flows by means of helicity. *AIAA Journal*, 28(8):1347–1352, August 1990.

[40] Lars Linsen, Valerio Pascucci, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Wavelet-based multiresolution with $\sqrt[n]{2}$ subdivision. *Journal on Computing*, special edition, 2004.

[41] Y. Livnat, Han-Wei Shen, and Christopher R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.

[42] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surfaces construction algorithm. In Maureen C. Stone, editor, *SIGGRAPH '87. Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–169, Anaheim, CA, USA, 27–31 July 1987. ACM SIGGRAPH, ACM Press.

[43] D. Lovely and Robert Haimes. Shock detection from computational fluid dynamics results. In *Proceedings of the 14th AIAA Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, 1999. AIAA paper 99-3285.

[44] Kwan-Liu Ma, John van Rosendale, and Willem Vermeer. 3D shock wave visualization on unstructured grids. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the 1996 Symposium on Volume Visualization*, pages 87–94,104, San Francisco, CA, USA, 28–29 October 1996. IEEE.

[45] K. W. Morton and M. A. Rudgyard. Shock recovery and the cell vertex scheme for the steady euler equations. In D. L. Dowyer, M. Yousuff Hussaini, and R. G. Voigt, editors, *Lecture notes in physics*, pages 424–428. Springer-Verlag, 1989.

[46] MPEG-1. ISO/IEC International Standard 11172.

[47] MPEG-2. ISO/IEC International Standard 13818.

[48] Hans-Georg Pagendarm and B. Seitz. An algorithm for detection and visualization of discontinuities in scientific data fields applied to flow data with shock waves. In P. Palamidese, editor, *Scientific Visualization: Advanced Software Techniques*, pages 161–177. Ellis Horwood Limited, 1993.

[49] Hans-Georg Pagendarm and Birgit Walter. Feature detection from vector quantities in a numerically simulated hypersonic flow field in combination with experimental flow visualization. In *Proceedings of the 5th Conference on Visualization '94*, pages 117–123, Washington, DC, USA, 17–21 October 1994. IEEE Computer Society, IEEE Computer Society Press.

[50] Valerio Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In Oliver Deussen, Charles D. Hansen, Daniel A. Keim, and Dietmar Saupe, editors, *Data Visualization 2004. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, Konstanz, Germany, 19–21 May 2004. Eurographics Association.

[51] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of Super Computing*, November 2001.

[52] Valerio Pascucci and Randall J. Frank. Hierarchical indexing for out-of-core access to multi-resolution data. In Gerald Farin, Bernd Hamann, and Hans Hagen, editors, *Hierarchical and Geometrical Methods in Scientific Visualization*, Mathematics and Visualization, pages 225–241. Springer-Verlag, 2003.

[53] Ronald Peikert and Martin Roth. The parallel vectors operator — a vector field visualization primitive. In David S. Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 10th Conference on Visualization '99*, pages 263–270, San Francisco, CA, USA, 24–29 October 1999. IEEE Computer Society, IEEE Computer Society Press.

[54] L. M. Portela. *On the Identification and Classification of Vortices*. PhD thesis, Stanford University, School of Mechanical Engineering, 1997.

[55] Frits H. Post and Jarke J. van Wijk. Visual representation of vector fields: Recent developments and research direction. In Lawrence Rosenblum, Rae A. Earnshaw, José Luis Encarnação, Hans Hagen, Arie E. Kaufman, S. V. Klimenko, Gregory M. Nielson, Frits H. Post, and Daniel Thalmann, editors, *Scientific Visualization: Advances and Challenges*, chapter 23, pages 367–390. Academic Press, London, 1994.

[56] Frits H. Post, Benjamin Vrolijk, Helwig Hauser, Robert S. Laramee, and Helmut Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, December 2003.

[57] Freek Reinders. *Feature-Based Visualization of Time-Dependent Data*. PhD thesis, Delft University of Technology, The Netherlands, March 2001.

[58] Freek Reinders, Melvin E. D. Jacobson, and Frits H. Post. Skeleton graph generation for feature shape description. In Willem C. de Leeuw and Robert van Liere, editors, *Data Visualization 2000. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 73–82, Amsterdam, The Netherlands, 29–31 May 2000. Springer-Verlag.

[59] Freek Reinders, Frits H. Post, and Hans J. W. Spoelder. Attribute-based feature tracking. In Eduard Gröller, Helwig Löffelmann, and W. Ribarsky, editors,

Data Visualization '99. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization, pages 63–72, Vienna, Austria, 26–28 May 1999. Springer-Verlag.

[60] Freek Reinders, Frits H. Post, and Hans J. W. Spoelder. Visualization of time-dependent data using feature tracking and event detection. *The Visual Computer*, 17(1):55–71, February 2001.

[61] Freek Reinders, I. Ari Sadarjoen, Benjamin Vrolijk, and Frits H. Post. Vortex tracking and visualisation in a flow past a tapered cylinder. *Computer Graphics Forum*, 21(4):675–682, November 2002.

[62] Freek Reinders, Hans J. W. Spoelder, and Frits H. Post. Experiments on the accuracy of feature extraction. In Dirk Bartz, editor, *Proceedings of the Ninth Eurographics Workshop on Visualization in Scientific Computing*, pages 49–58, Blaubeuren, Germany, 20–22 April 1998. Springer-Verlag.

[63] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[64] S. K. Robinson. Coherent motions in the turbulent boundary layer. *Annual Review of Fluid Mechanics*, 23:601–639, 1991.

[65] John van Rosendale. Floating shock fitting via lagrangian adaptive meshes. Technical Report 94-89, Institute for Computer Applications in Science and Engineering, 1994.

[66] Martin Roth. *Automatic Extraction of Vortex Core Lines and Other Line-Type Features for Scientific Visualization*. Diss. eth no. 13673, Swiss Federal Institute of Technology, ETH Zürich, 2000.

[67] Martin Roth and Ronald Peikert. Flow visualization for turbomachinery design. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the 7th Conference on Visualization '96*, pages 381–384, San Francisco, CA, USA, 27 October – 1 November 1996. IEEE Computer Society, IEEE Computer Society Press.

[68] Martin Roth and Ronald Peikert. A higher-order method for finding vortex core lines. In David S. Ebert, Holly Rushmeier, and Hans Hagen, editors, *Proceedings of the 9th Conference on Visualization '98*, pages 143–150, Research Triangle Park, NC, USA, 18–23 October 1998. IEEE Computer Society, IEEE Computer Society Press.

[69] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH '00. Proceedings of the 27th Annual*

Conference on Computer Graphics and Interactive Techniques, pages 343–352, New Orleans, LA, USA, 23–28 July 2000. ACM SIGGRAPH, ACM Press.

[70] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Proceedings of the 2001 Symposium on Interactive 3D graphics*, pages 63–68, New York, NY, USA, March 2001. ACM Press.

[71] I. Ari Sadarjoen and Frits H. Post. Geometric methods for vortex detection. In Eduard Gröller, Helwig Löffelmann, and W. Ribarsky, editors, *Data Visualization '99. Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 53–62, Vienna, Austria, 26–28 May 1999. Springer-Verlag.

[72] I. Ari Sadarjoen and Frits H. Post. Detection, quantification, and tracking of vortices using streamline geometry. *Computers & Graphics*, 24(3):333–341, June 2000.

[73] I. Ari Sadarjoen and Frits H. Post. Techniques and applications of deformable surfaces. In Hans Hagen, Gregory M. Nielson, and Frits H. Post, editors, *Scientific Visualization, Proceedings Dagstuhl '97*, pages 277–286. IEEE Computer Society, 2000.

[74] J. Sahm, I. Soetebier, and H. Birthelmer. Efficient representation and streaming of 3d scenes. *CANDG*, 28(1):15–24, February 2004.

[75] Ravi Samtaney, Deborah Silver, Norman J. Zabusky, and Jim Cao. Visualizing features and tracking their evolution. *IEEE Computer*, 27(7):20–27, July 1994.

[76] Gerik Scheuermann, Heinz Kruger, Martin Menzel, and Alyn P. Rockwood. Visualizing nonlinear vector field topology. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):109–116, April 1998.

[77] James A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2 edition, June 1999.

[78] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, 37(1):10–21, January 1949.

[79] Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In David S. Ebert, Holly Rushmeier, and Hans Hagen, editors, *Proceedings of the 9th Conference on Visualization '98*, pages 159–166, Research Triangle Park, NC, USA, 18–23 October 1998. IEEE Computer Society, IEEE Computer Society Press.

[80] Han-Wei Shen, L.-J. Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In David S. Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 10th Conference on Visualization '99*, pages 371–377,545, San Francisco, CA, USA, 24–29 October 1999. IEEE Computer Society, IEEE Computer Society Press.

[81] Han-Wei Shen, Charles D. Hansen, Y. Livnat, and Christopher R. Johnson. Isosurfacing in span space with utmost efficiency (issue). In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the 7th Conference on Visualization '96*, pages 287–294, San Francisco, CA, USA, 27 October – 1 November 1996. IEEE Computer Society, IEEE Computer Society Press.

[82] Claudio Silva, Yi-Jen Chiang, J. El-Sana, and Peter Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. Tutorial Course Notes, IEEE Visualization, 2002.

[83] Deborah Silver and Xin Wang. Tracking and visualizing turbulent 3D features. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):129–141, April 1997.

[84] Deborah Silver and Xin Wang. Visualizing evolving scalar phenomena. *Future Generation Computer Systems*, 15(1):99–108, February 1999.

[85] Deborah Silver, Norman J. Zabusky, V. Fernandez, M. Gao, and Ravi Samtaney. Ellipsoidal quantification of evolving phenomena. In N. M. Patrikalakis, editor, *Scientific Visualization of Natural Phenomena*, pages 573–588. Springer-Verlag, 1991.

[86] Simon Stegmaier, Ulrich Rist, and Thomas Ertl. Opening the can of worms: An exploration tool for vortical flows. In *Proceedings of the 16th Conference on Visualization 2005*, pages 463–470, Minneapolis, MN, USA, 23–28 October 2005. IEEE Computer Society, IEEE Computer Society Press.

[87] B. Stenger, P. R. S. Mendonça, and R. Cipolla. Model based 3D tracking of an articulated hand. In *Proc. Conf. Computer Vision and Pattern Recognition*, volume II, pages 310–315, Kauai, USA, December 2001.

[88] David Sujudi and Robert Haimes. Identification of swirling flow in 3D vector fields. AIAA Paper 95-1715, June 1995.

[89] Philip M. Sutton and Charles D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In David S. Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 10th Conference on Visualization '99*, pages 147–153,520, San Francisco, CA, USA, 24–29 October 1999. IEEE Computer Society, IEEE Computer Society Press.

[90] Xavier Tricoche, Gerik Scheuermann, and Hans Hagen. Continuous topology simplification of planar vector fields. In *Proceedings of the 12th Conference on Visualization 2001*, pages 159–166, San Diego, CA, USA, 21–26 October 2001. IEEE Computer Society, IEEE Computer Society Press.

[91] Xavier Tricoche, Thomas Wischgoll, Gerik Scheuermann, and Hans Hagen. Topology tracking for the visualization of time-dependent two-dimensional flows. *Computers & Graphics*, 26(2):249–257, April 2002.

[92] Edward Rolf Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

[93] Jayaram K. Udupa and Dewey Odhner. Shell rendering. *IEEE Computer Graphics and Applications*, 13(6):58–67, November 1993.

[94] Sander Pieter van der Pijl. *Computation of bubbly flows with a Mass-Conserving Level-Set method*. PhD thesis, Delft University of Technology, The Netherlands, 2005.

[95] John Villasenor and Alain Vincent. An algorithm for space recognition and time tracking of vorticity tubes in turbulence. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 55(1):27–35, January 1992.

[96] Benjamin Vrolijk, Charl P. Botha, and Frits H. Post. Fast time-dependent isosurface extraction and rendering. In Alexander Pasko, editor, *Proceedings of the 20th Spring Conference on Computer Graphics*, pages 45–54, Budmerice, Slovakia, 21–24 April 2004. ACM Press.

[97] Benjamin Vrolijk and Frits H. Post. Interactive out-of-core isosurface visualisation in time-varying data sets. *Computers & Graphics*, 30(2):265–276, April 2006.

[98] Theo van Walsum. *Selective Visualization on Curvilinear Grids*. PhD thesis, Delft University of Technology, The Netherlands, December 1995.

[99] Theo van Walsum, Frits H. Post, Deborah Silver, and Frank J. Post. Feature extraction and iconic visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):111–119, June 1996.

[100] Chaoli Wang and Han-Wei Shen. A framework for rendering large time-varying data using wavelet-based time-space partitioning (wtsp) tree. Technical Report OSU-CISRC-1/04-TR05, Computer and Information Science Research Center, Ohio State University, 2004.

[101] Chris Weigle and David C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Proceedings of the 1998 Symposium on Volume Visualization*, pages 103–110, Research Triangle Park, NC, USA, 19–20 October 1998. IEEE, ACM Press.

[102] Rüdiger Westermann. A multiresolution framework for volume rendering. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 51–58, Tysons Corner, VA, USA, 17–18 October 1994. ACM Press.

[103] Lee Westover. Interactive volume rendering. In *Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, Chapel Hill, NC, USA, May 1989. ACM Press.

[104] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[105] Lance J. Williams. Pyramidal parametrics. In Peter P. Tanner, editor, *SIGGRAPH '83. Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, pages 1–11, Detroit, MI, USA, 25–29 July 1983. ACM SIGGRAPH, ACM Press.

[106] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, , and Patricia Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 7–12, Boston, MA, USA, 28–29 October 2002. IEEE, ACM Press.

[107] Yang Yang. Multi-resolution visualization of large 4d data sets. Master's thesis, Delft University of Technology, The Netherlands, July 2005.

[108] L. A. Yates and G. T. Chapman. Streamlines, vorticity lines, and vortices. Technical Report AIAA-91-0731, American Institute of Aeronautics and Astronautics, 1991.

[109] Norman J. Zabusky, O. N. Boratav, R. B. Pelz, M. Gao, Deborah Silver, and S. P. Cooper. Emergence of coherent patterns of vortex stretching during reconnection: A scattering paradigm. *Physical Review Letters*, 67(18):2469–2472, 1991.

List of Figures

List of Tables

Summary

The research described in this thesis was part of a larger research project about multi-phase flows, in cooperation with researchers from Fluid Mechanics and Numerical Analysis. Multi-phase flows are characterised by a sharp transition between the fluids, the so-called phase front. There is no mixture between these fluids, therefore the transition from one fluid or phase to another is sharp and coincides with large jumps in physical quantities such as density and viscosity. One of the goals of the project was to study the evolution of the phase fronts using CFD, i.e. to study the development of the surfaces over time and to understand how they change and interact with each other. In order to study the evolving fronts, methods were needed for detecting and extracting them in the first place, and subsequently for tracking the phase fronts over time, and finding a way to visualise them interactively. Initially, feature extraction and tracking techniques were studied for this. However, after about a year into this project, it became apparent that the numerical method that was to be used for the simulation could provide an implicit surface representation of the phase front. This surface could easily be extracted from the data as an isosurface. Therefore, the research into feature extraction techniques was abandoned at that time and the focus of the research was redirected towards efficient techniques for interactive isosurfacing from very large time-dependent data sets.

Fast-access data structures that were designed to perform one particular visualisation task efficiently were examined first. These data structures make use of the properties of a particular visualisation algorithm and are made to fit the algorithm closely. One such data structure was explored in detail: the temporal index tree, in combination with the interval tree. The interval tree was designed for very fast isosurface cell

selection. This data structure is combined with a temporal index tree that makes use of data coherence in time to compress the data set in this dimension. The combined application of these data structures results in high speed isosurface cell extraction from time-dependent data sets. The extraction algorithm was combined with a special-purpose fast direct-rendering algorithm in order to prevent the rendering from becoming the bottleneck. The advantage of data structures like these is that they are designed to perform a particular visualisation task very fast. However, the drawback is that they are also limited to perform only that visualisation task. For extracting isosurfaces from time-dependent data sets, the index tree works very well, but the resulting visualisation cannot be combined with another type of visualisation. The data is stored in such a way in the index tree, that only isosurface cells can be extracted from it. The original, raw data set is not available, so the isosurface cannot be coloured with another scalar, or combined with streamlines, for example.

The second approach that was explored is the use of multi-resolution data structures. These structures enable the data to be accessed at several levels of resolution. The original data can be retrieved, but if available time or memory is limited, a lower resolution version of the data can also be requested. Whatever resolution is used, raw data is returned, which means that the visualisation is as flexible as in the original data. Unlike the fast-access data structures described above, (most) multi-resolution structures are not designed for one specific visualisation algorithm. This is a large difference between the two approaches, and a huge advantage of the latter. A multi-resolution data structure provides the flexibility to switch between different visualisations and is designed to handle large data sets by trading off data resolution for speed. There are many different ways to create a multi-resolution data structure. The method that was implemented in this project reorders the data in such a way that low resolution (subsampled) data is at the front of the data set and data points from consecutive higher levels of resolution are stored consecutively on disk. This has a number of advantages. First, there is no data replication, so the size of the multi-resolution data set is the same as that of the original data set; there is no spatial overhead. Secondly, data points from a single level of resolution are stored coherently on disk and therefore cache-friendly. Thirdly, because the data is merely reordered, lower resolution data is part of the higher resolution data. This means that the lower resolution data can be reused and less data has to be read when increasing the resolution. The fact that the low resolution data is only a subset of the high resolution data is also a disadvantage: if the data would be downsampled (filtered) instead of subsampled (unfiltered), less information would be lost in the lower levels of resolution, resulting in the data and visualisation being more smooth. The multi-resolution approach was extended to time-dependent data sets. Techniques for region-of-interest selection and time-window management were added to provide interactive visualisation and space-time navigation of these large 4D data sets.

Samenvatting

Het onderzoek dat in dit proefschrift is beschreven, was een deel van een groter onderzoeksproject over meerfasenstromingen, in samenwerking met onderzoekers van Stromingsleer en Numerieke Analyse. Meerfasenstromingen worden gekarakteriseerd door een scherpe overgang tussen de (vloei)stoffen, het zogenaamde fasefront. Er treedt geen vermenging op tussen de stoffen, daardoor is de overgang van de ene (vloei)stof of fase naar de andere scherp en valt deze samen met grote sprongen in fysieke grootheden zoals dichtheid en viscositeit. Een van de doelen van het project was om de evolutie van de fasefronten te bestuderen met behulp van CFD, oftewel om de ontwikkeling van de oppervlakken in de tijd te bestuderen en te begrijpen hoe deze veranderen en elkaar beïnvloeden. Om de evoluerende fasefronten te bestuderen, waren methoden nodig om ze in de eerste plaats te detecteren en te extraheren, vervolgens om ze te kunnen volgen (*tracken*) in de tijd en tenslotte om de fronten interactief te kunnen visualiseren. Aanvankelijk zijn hiervoor feature-extractie- en -trackingtechnieken bestudeerd. Echter, na ongeveer een jaar werd het binnen het project duidelijk, dat de numerieke methode die gebruikt zou gaan worden voor de simulatie een impliciete oppervlakterepresentatie van het fasefront kon aanleveren. Dit oppervlak kon eenvoudig als een iso-oppervlak uit de data geëxtraheerd worden. Daarom is op dat moment het onderzoek naar feature-extractietechnieken afgebroken en is de aandacht van het onderzoek gericht op efficiënte technieken voor interactieve extractie van iso-oppervlakken uit zeer grote tijdsafhankelijke datasets.

Eerst is gekeken naar datastructuren voor snelle toegang, die ontworpen zijn om één bepaalde visualisatietaak efficiënt uit te voeren. Deze datastructuren maken gebruik van de eigenschappen van een bepaald visualisatiealgoritme en zijn erop gemaakt om

nauw bij dit algoritme aan te sluiten. Eén zo'n datastructuur is in detail onderzocht: de *temporal index tree* in combinatie met de *interval tree*. De interval tree is ontworpen voor zeer snelle selectie van iso-oppervlakcellen. Deze datastructuur wordt gecombineerd met een temporal index tree die gebruik maakt van de coherentie van data in de tijd om de dataset in deze dimensie te comprimeren. De gecombineerde toepassing van deze datastructuren resulteert in zeer snelle extractie van iso-oppervlakcellen uit tijdsafhankelijke datasets. Het extractiealgoritme is gecombineerd met een specifiek voor dit doel gemaakt *direct rendering* algoritme, om te voorkomen dat de rendering de bottleneck zou worden. Het voordeel van dergelijke datastructuren is dat deze zijn ontworpen om een bepaalde visualisatietaak zeer snel uit te voeren. Echter, het nadeel is dat ze ook beperkt zijn tot slechts die visualisatietaak. De index tree werkt zeer goed om iso-oppervlakken te extraheren uit tijdsafhankelijke datasets, maar de resulterende visualisatie kan niet gecombineerd worden met een ander type visualisatie. De data zijn op zodanige wijze opgeslagen in de index tree, dat alleen iso-oppervlakcellen eruit geëxtraheerd kunnen worden. De originele, ruwe dataset is niet beschikbaar, dus het iso-oppervlak kan bijvoorbeeld niet gekleurd worden met een andere scalaire waarde, of gecombineerd worden met stroomlijnen.

De tweede aanpak die is onderzocht is het gebruik van multiresolutie-datastructuren. Deze structuren bieden de mogelijkheid om de data op meerdere resolutieniveaus te benaderen. De originele data kunnen opgevraagd worden, maar als de beschikbare hoeveelheid tijd of geheugen beperkt is, is het ook mogelijk de data in een lagere resolutie op te vragen. Welke resolutie ook gebruikt wordt, er worden altijd ruwe data geretourneerd, wat betekent dat de visualisatie net zo flexibel is als in de originele data. In tegenstelling tot de datastructuren voor snelle toegang, die hierboven werden beschreven, zijn (de meeste) multiresolutie-structuren niet ontworpen voor één specifiek visualisatiealgoritme. Dit is een belangrijk verschil tussen deze twee aanpakken en een groot voordeel van laatstgenoemde. Een multiresolutie-datastructuur biedt de flexibiliteit om te schakelen tussen verschillende visualisaties en is ontworpen om grote datasets te kunnen hanteren door resolutie in te wisselen voor snelheid. Er zijn vele manieren om een multiresolutie-datastructuur te maken. De methode die in dit project is geïmplementeerd, herordent de data op zodanige wijze dat lage-resolutie (gesubsamplede) data aan het begin van de dataset komen en datapunten van de achtereenvolgende hogere resolutieniveaus na elkaar worden opgeslagen op schijf. Dit heeft een aantal voordelen. Ten eerste is er geen datareplicatie, dus de grootte van de multiresolutie-dataset is hetzelfde als die van de originele dataset; er is geen ruimtelijke overhead. In de tweede plaats worden punten uit één resolutieniveau coherent en daardoor *cache-friendly* op schijf opgeslagen. In de derde plaats, omdat de data slechts worden herordend, maken de lagere-resolutie data deel uit van de hogere-resolutie data. Dit betekent dat de lagere-resolutie data hergebruikt kunnen worden en dat minder data ingelezen hoeven te worden, wanneer de resolutie verhoogd wordt. Het feit dat de lage-resolutie data slechts een subset zijn van de hoge-resolutie data is ook een nadeel: als de data zouden worden gedownsampled (gefilterd) in

plaats van gesubsampled (ongefilterd), zou minder informatie verloren gaan in de lagere resolutieniveaus, hetgeen zou resulteren in gladdere data en visualisaties. Deze multiresolutie-aanpak is uitgebreid naar tijdsafhankelijke datasets. Technieken voor de selectie van een aandachtsgebied (*region of interest*) en voor het beheren van het tijdvenster zijn toegevoegd om interactieve visualisatie van en ruimte-tijd-navigatie in deze grote 4D datasets mogelijk te maken.

Curriculum Vitae

Benjamin Vrolijk was born on January 27th, 1978, in Katwijk aan den Rijn, The Netherlands. At the age of 4 he moved to Leeuwarden, where he started grammar school at the Christelijk Gymnasium in 1989. After moving to Capelle aan den IJssel in 1991, he finished grammar school at the Comenius College in 1995. He started studying Computer Science at the Delft University of Technology in September of the same year. In September 2001 he received his Master's Degree after doing a project on feature tracking in the Computer Graphics and CAD/CAM group. Continuing in the same group and research field, he started his PhD project on September 1st, 2001. This project was about the visualisation of surfaces in very large, time-dependent data sets, as part of a larger research project into multi-phase flows. Since February 2006 he is working at Kinsley bv, in Breda, as a software engineer in (web-based) database applications.