Acceleration Techniques for Numerical Flow Visualization

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Simon Stegmaier

aus Mutlangen

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Ir. F. H. Post
Tag der mündlichen Prüfung:	13. Juni 2006

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2006

Dedicated to my mother
who did not stop supporting me
despite all her sorrows.

Contents

List of Abbreviations and Acronyms

API	Application Program Interface
bit	binary digit
bzgl.	bezüglich
bzw.	beziehungsweise
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CT	Computed Tomography
d.h.	das heißt
DMA	Direct Memory Access
DNS	Direct Numerical Simulation
Dr. rer. nat.	Doctor rerum naturalium
e.g.	exempli gratia
etc.	et cetera
fps	frames per second
GB	gigabyte
GPU	Graphics Processing Unit
GUI	Graphical User Interface
Hz	Hertz
i.e.	id est
Ir.	Ingenieur
KB	kilobyte
KBps	kilobytes per second
kHz	kilohertz
LAN	Local Area Network
LDA	Laser Doppler Anemometry
LES	Large Eddy Simulation
LGA	Lattice Gas Automata
LIC	Line Integral Convolution
MB	megabyte
Mbps	megabits per second
MC	Marching Cubes
MRI	Magnetic Resonance Imaging
MRT	Magnetic Resonance Tomography
ODE	Ordinary Differential Equation
PC	Personal Computer
PDA	Personal Digital Assistant
PIV	Particle Image Velocimetry
pixel	picture element
RANS	Reynolds-Averaged Navier-Stokes
RGB	red, green, blue
RGBA	red, green, blue, alpha
s	second
SIMD	Single Instruction, Multiple Data
sog.	sogenannte
u.a.	unter anderem
WLAN	Wireless LAN

Abstract and Chapter Summaries (in English and German)

Abstract

This thesis addresses the problem of making computer-aided flow visualization more efficient and more effective. More efficient because several new algorithms are presented for accelerating the visualization itself; more effective because accelerated visualization yields more productive work during data analysis.

Whether there is a need for acceleration techniques depends on several parameters. Obviously, there is a strong dependence on the available computing hardware: what is reasonable on one hardware platform might be unbearable on another platform. This straightforwardly leads to the idea of switching to another (remote) visualization platform while keeping the researcher's workspace untouched. Alternatively, more efficient use of local hardware resources can be made, a direction followed in this thesis by balancing the workload between the (programmable) graphics hardware and the central processing unit. Instead of exploiting parallel processing, reduced accuracy can be traded for improved interactivity. In this work, this trade-off is made by converting the grid underlying the data to a representation that can be handled more efficiently. In the worst case, neither hardware approaches nor accuracy reduction sufficiently improve the data analysis. Consequently, data reduction must be employed to keep up with human cognition capabilities and limited graphics processing resources. This issue is addressed by flow feature extraction which aims at presenting a highly compact representation of the data.

This work thus presents a unique multi-level approach for accelerating flow visualization, considering hardware resources, accuracy requirements, and cognitive issues. Due to the generality of the selected acceleration techniques presented in this thesis, some results do also have impact on other areas of scientific visualization. Furthermore, due to the layered approach addressing the acceleration on multiple abstraction levels, the presented techniques can be used stand-alone as well as in combination to yield a highly flexible toolbox that can be fine-tuned to the respective environment.

Chapter Summaries

This section gives an overview of this thesis by means of chapter summaries.

Chapter 1: Introduction

This chapter is intended to introduce the thesis topic. We will argue that flows are important in many research fields and that many techniques have evolved for simulating and measuring flows. Increasing data set sizes will be identified as the major factor making the data analysis increasingly more difficult.

Since this applies to many research fields, we will elaborate on why flow visualization is special in this respect. For this purpose we will discuss the high diversity of data set sizes found in practice, the impact of available hardware, and the importance of the visualization task for the question of what the term “large data” actually translates to.

This discussion will define the need for a problem solution on several levels taking into account various parameters. The chapter closes with a discussion of the benefits of the approach chosen for this work, which will be identified to be free choice of the level of interference involved in the acceleration techniques with regard to data, visualization techniques, and applications.

Chapter 2: Graphics and Visualization Techniques

This chapter introduces the technologies and techniques used for this thesis. We start by introducing the visualization pipeline describing the transformations from the raw data to the final image. Then we introduce the various kinds of grids used for the data analysis and which define data values at discrete points in space. We will next elaborate on interpolation in grids, i.e. determining values at arbitrary (off-grid) locations in data sets given at discrete points.

What follows is an overview of flow visualization. We will start by giving a formal definition of flows and we will discuss the distinction between physical flow visualization and numerical flow visualization as addressed in this thesis. For physical flow visualization we will discuss techniques like streak lines, path lines, and time lines, LDA, PIV, shadowgraphs and Schlieren photography. Although none of these techniques is used in this work (only indirectly as data provider) they motivate several numerical flow visualization techniques described next.

For numerical flow visualization we will explain how to derive stream lines numerically and then give short overviews of other flow visualization techniques like dense vector field visualization techniques (LIC), cutting planes, and color encodings.

The next section discusses feature-based flow visualization and in particular the detection of vortices contained in a flow field. For this purpose, we will first define the term vortex and then examine algorithms based on vorticity. We then discuss two advanced vortex detection methods, namely predictor-corrector approaches utilizing pressure and vorticity, and the λ_2 method widely used in this work which is based on a decomposition of the velocity gradient tensor.

We will continue with an overview of volume visualization used for visualizing scalar fields associated with a flow (pressure, density, λ_2 values). A differentiation between direct volume visualization and indirect volume visualization is then given before elaborating on the visualization techniques in detail.

For direct volume visualization we will first give a derivation of the volume rendering integral and then give descriptions of two popular volume rendering approaches: slice-based volume rendering and ray-casting. For both approaches we will discuss the image compositing and their pros and cons.

We then switch to indirect volume visualization using isosurfaces. This discussion is restricted to isosurface extraction using the *Marching Cubes* algorithm.

The remaining part of the chapter discusses technical issues at a lower abstraction level. We start by giving an overview of the rendering pipeline that transforms vertices, lines, and polygons to pixels. Since this is all accomplished by the graphics hardware we will discuss graphics processing units next. A major issue in this discussion will be the use of graphics hardware for general-purpose computations. For this we will elaborate on both vertex and fragment programs and the corresponding processing units.

Once a computer-generated image has been created is has to be displayed. We discuss the use of various stereo techniques (active/passive, ChromaDepth) to provide depth cues and explain how frusta required for the stereo pair are derived.

For the final display we give an overview of the X Window system architecture, the associated client/server model and the programming alternatives. Of special interest will be the integration of graphics libraries and the differentiation between direct and indirect rendering.

The last topic discussed in the chapter is intended to brush up the concepts of dynamic linking and loading, techniques for providing software library functionality which are extensively used in this work to obtain generic behavior.

Chapter 3: Remote Visualization

This chapter presents novel approaches for remote visualization. We will argue that increasingly large data sets (mainly from numerical simulations) often cannot be handled by local hardware available at the researcher's desk and that there are alternatives for visualizing the data as long as there are remotely located computing and graphics resources accessible via a fast network.

For this purpose we will first discuss previous solutions and the extent to which they differ from the novel approaches devised for this work. This will include discussions of the *Visapult* system and the *Vizserver*.

Following this we will discuss two alternative, novel architectures. The first architecture exploits X Window mechanisms for transmitting image data and is completely generic in the sense that no source code modifications at all will be required for adapting the application to be used remotely. We will discuss first the basic architecture and then elaborate on implementation details like selecting trigger functions which indicate the termination of the rendering and, accordingly, signal clearance for image transmission.

Based on this basic architecture we will then discuss methods for improving the system performance using low-bandwidth X servers and the VNC remote desktop tool.

A second alternative for the system architecture is discussed next. For deriving the basic idea behind this architecture we will examine weaknesses and strengths of the original architecture and subsequently develop a revised architecture based on dedicated network channels for image transmission.

As before, optimizations yielding better system performance are elaborated on in the following. The first optimizations address data reduction using custom image compression. Comparisons will be given for different compression algorithms. Then we will elaborate on methods to improve interactivity by employing user-interaction–controlled downsampling. We will proceed by examining methods for reducing latencies. Two approaches will be presented: first, parallel processing balancing GPU- and CPU-intensive work and, second, image compression utilizing graphics hardware. All optimizations are critically evaluated.

The remaining chapter sections elaborate on how remote visualization can be made more useful. For this purpose we will first describe a method for generically adapting user interfaces for small-screen devices like PDAs. This is required functionality if also handheld devices are to be used for remote visualization (which the presented remote visualization solutions in principle allow for). We begin our discussion by defining criteria for a handheld-suitable GUI by examining possible user interactions and the impact of reduced screen space on user interfaces designed for desktop computers. Then we will present a novel approach for adapting the GLUT toolkit to meet the defined criteria.

We close the chapter with a discussion of how to integrate stereo capabilities into a remote visualization system. It will be argued that anaglyphs are the most suitable technology for our application and we will give details on how to implement anaglyph stereo in general. Subsequently, we will give a detailed explanation of how to derive accurate stereo frusta for a given monoscopic view—again without accessing the application source code. Since the scene needs to be rendered twice we will next discuss alternatives for enforcing redrawings and we will iden-

tify event-based redrawing as the method of choice. A performance evaluation will demonstrate that the given implementation achieves near-optimal framerates. The discussion closes with the description of a stand-alone configuration tool for adjusting stereo parameters.

Chapter 4: Grid Resampling

In this chapter we introduce grid resampling techniques aiming at converting the data set structure to a format that can be processed more efficiently. We will argue that the resulting performance benefit may eliminate the need for remote visualization solutions under certain circumstances and also that Cartesian grids and hierarchies of Cartesian grids present a good balance between accuracy and improved performance. We will then elaborate on an implementation of a grid resampling tool devised for this thesis which has been designed for real-world applications in the car manufacturing industry.

We start our discussion by giving a short overview of various flow simulation packages and the grid types resulting from these simulations to further motivate the choice of hierarchical Cartesian grids. The discussions will also illustrate that efficient algorithms for this type of grid are readily available in commercial flow visualization packages and that resampling paves the road for high-performance visualizations of many data sets otherwise hard to analyze.

Next we will examine existing remote visualization solutions and discuss their pros and cons. Two alternative resampling approaches will be developed in the following: semi-automatic resampling and fully-automatic, adaptive resampling.

Semi-automatic resampling requires the user to manually define refinement regions. An evaluation of this approach will show that this object-based approach is fast and straightforward to implement but also that it may be inefficient with regard to the number of grid cells and the resulting accuracy, thereby motivating the need for adaptive resampling.

We will argue that a geometry-based approach for adaptive resampling is to be preferred over an error-based approach from a performance point of view and we will identify three major phases involved in the resampling process: construction of an octree, octree correction, and interpolation. For the octree construction we will argue that standard octrees are inefficient and intolerable for practical data set sizes. We will then give notations and definitions for a memory-efficient pointerless octree implementation and elaborate on standard operations required for accessing data. The need for octree correction will be motivated by the lack of visualization functionality for unrestricted octrees in existing flow visualization software. And algorithms will be given for efficiently performing this correction step. Finally, we will elaborate on the interpolation phase and discuss a parallel implementation of the adaptive resampling approach.

The chapter closes with a comparative evaluation of the presented resampling algorithms with respect to processing performance and attained accuracy in respect of the number of generated cells.

Chapter 5: Feature Extraction

This chapter addresses issues related to the extraction of flow features. We will argue that visualizing flow features is beneficial due to an inherent data reduction and easily to interpret results. The discussion will also show that the respective algorithms are computationally expensive.

We will start by discussing flow simulation methods in terms of noise introduced into the flow field and we will illustrate that both experimental as well as numerical methods are vulnerable to noise. The discussion will cover methods like LDA, PIV, DNS, LES, and RANS. This will be used to motivate the need for filtering and denoising and the need for fast feature extraction algorithms and fast vortex detection methods in particular.

We proceed by giving an optimized software implementation of the λ_2 vortex detection method. For this purpose we will elaborate on how to efficiently compute velocity gradient tensors and on how to efficiently compute eigenvalues. The implementation will be evaluated showing that interactive vortex extraction is impossible for typical data sets found in practice.

Therefore, we will next give a GPU-based implementation of the λ_2 method which covers denoising, vortex detection, and the final visualization. We will discuss in detail the selection of filter kernels and the implementation of separated filters and we present strategies for porting the eigenvalue computation to the GPU. For visualization we will review various volume visualization approaches and eventually devise a volume visualization technique providing a good trade-off between quality and performance and which is applicable without interference of the CPU. We will evaluate the presented implementation and compare it with the software implementation, thereby showing that interactive work becomes possible.

Since vortices are often visualized by means of isosurfaces we will then proceed by examining GPU-accelerated extraction of isosurfaces. For this purpose we describe a GPU-based implementation of the *Marching Tetrahedra* algorithm capable of extracting isosurface geometry (which then can be post-processed) and a GPU-based implementation of volume ray-casting. The latter will be shown not to be restricted to isosurfaces which we will prove by presenting a series of other advanced direct and indirect volume visualization shaders. Both approaches are critically evaluated.

The chapter closes with a novel approach for segmenting vortices which allows for filtering on a vortex basis and which, thus, accelerates the flow visualiza-

tion itself as well as the data analysis since the researcher is enabled to concentrate on structures relevant for the respective task.

We first discuss related approaches to illustrate the need for improved segmentation capabilities and then identify predictor-corrector methods and the λ_2 method as ideal counterparts for reliable vortex detection and segmentation. We will give a detailed account of the implementation and evaluate the segmentation results using real-world data.

We then proceed by examining the needs for advanced interaction possibilities with regard to the segmented vortex structures and describe the design and implementation of a vortex browser allowing for picking, hiding, filtering, and manipulating vortices. The system is complemented by standard flow visualization techniques to visualize the vortex context, i.e. the enclosing vector field. The techniques includes cutplanes, hedgehog visualization, LIC, color encodings, particle traces, and isosurfaces of various associated scalar fields. Stereo visualization is provided to facilitate understanding complex interwoven structures.

The discussion closes with an evaluation of the system performed by practitioners.

Chapter 6: Conclusion

The last chapter resumes the algorithms and techniques developed for this thesis. To aid in selecting a suitable acceleration technique for the respective case we will give the conclusion by means of a flow diagram which yields a recommended approach based on a number of decisions.

The thesis closes with an outlook and directions for future work.

Zusammenfassung

Diese Dissertation befasst sich mit mit der Verbesserung von Effizienz und Effektivität computerunterstützter Strömungsvisualisierung. Erhöhte Effizienz wird dabei durch mehrere neue, im Folgenden vorgestellte Algorithmen erreicht, die die Visualisierung an sich beschleunigen, höhere Effektivität dadurch, dass beschleunigte Visualisierung die Datenanalyse produktiver gestaltet.

Der Bedarf für die Beschleunigung von Visualisierungstechniken hängt von mehreren Parametern ab. Offensichtlich existiert zunächst einmal eine starke Abhängigkeit von den zur Verfügung stehenden Ressourcen: eine Visualisierungstechnik, die auf einer Hardwareplattform vernünftig eingesetzt werden kann, mag auf anderen Plattformen aufgrund fehlender Rechenleistung völlig unbrauchbar sein. Dies führt geradewegs zu der Idee, für die Visualisierung entfernte Rechnerressourcen einzusetzen, dabei jedoch die lokal vorhandene Hardware am Ar-

beitsplatz des Benutzers unberührt zu lassen. Alternativ können die zur Verfügung stehenden Hardwareressourcen auch effizienter eingesetzt werden. Diese Richtung wird in dieser Arbeit durch Lastbalancierung zwischen (programmierbarer) Graphikhardware und der Zentraleinheit des Rechners verfolgt. Anstelle der Ausnutzung paralleler Verarbeitung kann auch eine Reduzierung der Datengenauigkeit treten, sofern hierdurch höhere Interaktivität erreicht wird. Dieser Kompromiss wird in dieser Arbeit anhand von Gitterkonvertierungsverfahren diskutiert, die den Daten zugrundeliegende Gitter in eine effizienter zu handhabende Form überführen. Im schlimmsten Falle reichen jedoch weder Hardwareansätze noch Genauigkeitsreduzierungen aus, um die Datenanalyse ausreichend zu verbessern. Folgerichtig muss demnach eine Datenreduktion durchgeführt werden, die die ursprüngliche Datenmenge auf ein Maß reduziert, das eine Verarbeitung durch die menschliche Wahrnehmung und begrenzte Graphikressourcen ermöglicht.

In dieser Dissertation wird daher ein neuartiger Ansatz verfolgt, der die Beschleunigung der Strömungsvisualisierung auf mehreren Ebenen unter Berücksichtigung zur Verfügung stehender Hardwareressourcen, etwaiger Genauigkeitsanforderungen und menschlicher Wahrnehmungsfähigkeiten behandelt. Aufgrund der Allgemeingültigkeit einiger gewählter Beschleunigungstechniken sind die erzielten Ergebnisse teils auch für andere Bereiche wissenschaftlicher Visualisierung von Bedeutung. Da weiterhin ein Schichtenmodell gewählt wurde, können die vorgestellten Beschleunigungstechniken sowohl separat als auch in Kombination verwendet werden. Hierdurch wird es dem Forscher möglich, Art und Anzahl der eingesetzten Techniken auf die individuellen Bedürfnisse und Rahmenbedingungen abzustimmen.

Kapitelzusammenfassungen

Die folgenden Abschnitte geben einen Überblick über diese Arbeit in Form von Kapitelzusammenfassungen.

Kapitel 1: Einleitung

Dieses Kapitel führt in das Thema der Dissertation ein. Wir werden argumentieren, dass Strömungen für viele Gebiete der Wissenschaft von großer Bedeutung sind, und dass eine Vielzahl unterschiedlichster Verfahren zur Berechnung und Messung von Strömungen entwickelt wurde. Zunehmend umfangreichere Datenmengen werden als Hauptursache zunehmend schwierigerer Datenanalyse identifiziert werden.

Da dies auf viele Forschungsgebiete zutrifft, werden wir im Folgenden darauf eingehen, warum Strömungen eine Sonderrolle einnehmen, und weshalb sich die

Problemstellung von denen anderer Bereiche unterscheidet. Zu diesem Zwecke werden wir auf die Größenunterschiede in der Praxis auftretender Datensätze eingehen, den Einfluss verfügbarer Hardwareressourcen diskutieren und die Bedeutung der Visualisierungsaufgabe für die Beantwortung der Frage, was letztendlich unter „großen Datenmengen“ zu verstehen ist, besprechen.

Diese Diskussion wird zeigen, dass eine adäquate Problemlösung eine Behandlung auf mehreren Ebenen unter Berücksichtigung verschiedener Parameter erfordert. Das Kapitel schließt mit einer Diskussion der Vorzüge, die durch den gewählten Ansatz gewonnen werden. Wir werden den Hauptvorteil darin identifizieren, dass der Benutzer sehr genau vorgeben kann, in welchem Maße in die Daten und die eingesetzten Visualisierungstechniken und -anwendungen eingegriffen wird.

Kapitel 2: Graphik- und Visualisierungstechniken

Dieses Kapitel führt in die für die Arbeit relevanten Technologien und Techniken ein. Wir beginnen mit einer Einführung der sog. *Visualisierungspipeline*, die die Transformation der Rohdaten bis zum Endbild beschreibt. Dann werden wir die verschiedenen Gittertypen zur Speicherung diskreter Datenpunkte einführen und auf die Interpolation von Datenwerten zwischen Gitterpunkten eingehen.

Hieran schließt sich ein Überblick über Strömungsvisualisierung an. Wir werden mit einer formalen Definition von Strömungen beginnen und anschließend physikalische Strömungsvisualisierung von der in dieser Arbeit behandelten numerischen Strömungsvisualisierung abgrenzen. Zur physikalischen Strömungsvisualisierung werden wir Streichlinien, Pfadlinien und Zeitlinien, LDA und PIV sowie Schattenverfahren und Schlierenphotographie besprechen. Obwohl keine dieser Techniken in dieser Arbeit eingesetzt wurde – lediglich indirekt als Datenlieferanten – motivieren sie dennoch einige Visualisierungsverfahren der numerischen Strömungsvisualisierung.

Für die numerische Strömungsvisualisierung wiederum werden wir erläutern, wie Strömungslinien numerisch berechnet werden, und wir werden eine kurze Übersicht über andere Visualisierungstechniken geben, die in dieser Dissertation zum Einsatz kamen. Hierzu zählen Methoden zur dichten Vektorfeldvisualisierung (LIC), Schnittebenen und Farbkodierungen.

Das nächste Unterkapitel diskutiert Merkmals-basierte Strömungsvisualisierung und insbesondere die Erkennung von in einer Strömung enthaltenen Wirbeln. Zu diesem Zwecke werden wir zunächst den Begriff „Wirbel“ definieren und anschließend auf der Wirbelstärke basierende Algorithmen zur Wirbelerkennung genauer betrachten. Im Anschluss hieran werden wir zwei verbesserte Verfahren zur Erkennung von Wirbeln besprechen: sog. *Predictor-Corrector* Verfahren, die außer der Wirbelstärke auch noch Druck zu Rate ziehen, und die sog. λ_2 Metho-

de, die auf der Berechnung des Geschwindigkeitsgradiententensors basiert und an mehreren Stellen in dieser Arbeit Verwendung finden wird.

Wir werden fortfahren mit einer Beschreibung von Verfahren zur Visualisierung von Volumendaten. Die Verfahren werden verwendet werden, um die zu einer Strömung gehörigen Skalardatenfelder wie Druck, Dichte oder λ_2 zu visualisieren. Wir werden dann direkte von indirekter Volumenvisualisierung abgrenzen und anschließend Algorithmen aus beiden Bereichen näher besprechen.

Zur direkten Volumenvisualisierung werden wir zunächst das Volumenrenderingintegral herleiten und dann zwei weit verbreitete Verfahren zur Darstellung von Volumen näher erläutern: texturbasierte Visualisierung und Strahlverfolgung. Für beide Ansätze werden wir die Bildzusammensetzung diskutieren und auf Vor- und Nachteile eingehen.

Wir werden dann mit einer Beschreibung indirekter Volumenvisualisierung mittels Isoflächen fortfahren. Dieser Teil wird sich auf die Extraktion von Isoflächengeometrie mit dem sog. *Marching Cubes* Algorithmus beschränken.

Der Rest des Kapitels widmet sich technischen Themen auf einem niedrigeren Abstraktionsniveau. Wir beginnen mit einem Überblick über die sog. *Renderingpipeline*, die Punkte, Linien und Polygone in Pixel transformiert. Da all dies in Graphikhardware geschieht, werden wir uns anschließend mit dieser Hardwareressource näher befassen. Das Hauptaugenmerk der Diskussion wird dabei auf der Verwendung von Graphikhardware für allgemeine (d.h. nicht Graphik-bezogene) Berechnungen liegen. Zu diesem Zwecke werden wir genauer auf die Geometrie- und Rasterisierungseinheit und die Programmierung dieser Einheiten eingehen.

Ein einmal fertig gestelltes Computer-generiertes Bild muss dargestellt werden. Dies kann mit Hilfe verschiedener Stereotechniken erfolgen, um beim Betrachter Tiefeneindruck zu erzeugen. Wir werden sowohl aktive als auch passive Techniken zur Erzeugung dieses Tiefeneindrucks besprechen und mit dem sog. *ChromaDepth* Verfahren auch ein farbbasiertes Verfahren kennenlernen. Im Anschluss hieran werden wir erklären, wie die beiden benötigten Stereoansichten berechnet werden.

Zur endgültigen Darstellung des Bildes werden wir dann auf die Architektur des X Window Fenstersystems eingehen, dessen Client/Server Modell und die vorhandenen Möglichkeiten der Programmierung. Von besonderem Interesse werden hierbei die Integration von Graphikbibliotheken und die Möglichkeiten der direkten und indirekten Graphikausgabe sein.

Der letzte Abschnitt des Kapitels erläutert die Konzepte des dynamischen Bindens und Ladens, also Techniken, die es erlauben, Anwendungen die Funktionalität von Softwarebibliotheken zur Verfügung zu stellen. Beide Konzepte werden in dieser Arbeit intensiv genutzt werden.

Kapitel 3: Fernvisualisierung

Dieses Kapitel stellt einen neuen Ansatz zur Fernvisualisierung vor. Wir werden argumentieren, dass zunehmend größere Datensätze (hauptsächlich von numerischen Simulationen) häufig nicht befriedigend mit den am Arbeitsplatz zur Verfügung stehenden Ressourcen analysiert werden können, dass andererseits aber Alternativen bestehen, solange entfernte Graphikressourcen und ein schnelles Verbindungsnetzwerk zur Verfügung stehen.

Zu diesem Zwecke werden wir zunächst verwandte Arbeiten besprechen und darauf eingehen, inwiefern sich diese von den vorgeschlagenen Lösungen unterscheiden. Die Diskussion wird hierbei u.a. auf das *Visapult* System und den *Vizserver* eingehen.

Im Anschluss hieran werden wir zwei alternative, neue Architekturen zur Fernvisualisierung vorstellen. Die erste Lösung greift auf Mechanismen des X Window Fenstersystems zur Übertragung der Bilddaten zurück und ist vollständig generisch in dem Sinne, dass keinerlei Änderungen am Quelltext der Applikation erforderlich sind, um diese zur Fernvisualisierung einsetzen zu können. Wir werden zunächst die grundsätzliche Architektur dieses System vorstellen und anschließend auf Implementierungsdetails genauer eingehen, beispielsweise die Auswahl von Auslöserfunktionen, die das Ende der Bilderzeugung signalisieren und damit die Freigabe zur Übertragung der Bilddaten geben.

Basierend auf dieser Architektur werden wir dann Verfahren besprechen, um die Systemleistung durch Einsatz spezieller X Server für Netzwerke niedriger Bandbreiten und des Softwarepakets *VNC* zur Fernadministration von Rechnersystemen zu beschleunigen.

Eine zweite Alternative wird im Folgenden vorgestellt werden. Zur Motivation dieser Architektur werden wir zunächst die Stärken und Schwächen der Basisarchitektur identifizieren und im Anschluss eine neue Architektur entwerfen, die zur Übertragung der Bilddaten eine gesonderte Datenleitung verwendet.

Wie zuvor werden wir auch für die überarbeitete Lösung Optimierungen beschreiben. Die zuerst besprochenen Optimierungen sorgen für eine Datenreduktion durch Komprimierung. Mehrere Komprimierungsverfahren werden hierbei verglichen werden. Wir werden dann eine Methode besprechen, die durch interaktionsgesteuerte Auflösungsreduzierung die Interaktivität der Anwendung erhöht. Hieran schließt sich eine Vorstellung zweier Methoden zur Latenzreduzierung an: parallele Bearbeitung mit Lastbalancierung zwischen der CPU und der GPU einerseits und Bildkomprimierung unter Ausnutzung von Graphikhardware andererseits. Alle Optimierungen werden kritisch beurteilt werden.

Die restlichen Unterkapitel diskutieren, wie der Wert von Fernvisualisierung erhöht werden kann. Hierzu werden wir zunächst ein Verfahren vorstellen, das es erlaubt, für Bildschirme normaler Größe entworfene Benutzungsoberflächen für

Kleinstgeräte anzupassen. Diese Funktionalität ist unabdingbar, falls auch mobile Geräte zur Fernvisualisierung eingesetzt werden sollen (was die vorgeschlagenen Lösungen grundsätzlich ermöglichen). Die Diskussion wird sich daher zunächst der Definition von Kriterien widmen, welche eine für Kleinstcomputer geeignete Benutzungsoberfläche erfüllen sollte. In diesem Zusammenhang werden die Auswirkungen eingeschränkter Benutzerinteraktionen und geringer Anzeigauflösungen besprochen werden. Ausgehend von diesen Kriterien werden wir dann einen neuartigen Ansatz vorstellen, um die Funktionalität von GLUT, einer vom Fenstersystem unabhängigen Bibliothek zur Erstellung von Benutzungsoberflächen, anzupassen.

Das Kapitel schließt mit einer Diskussion darüber, wie Stereovisualisierung und Fernvisualisierung vereint werden können. Wir werden darlegen, dass Anaglyphen die geeignetste Technologie darstellen, und wir werden erklären, wie Anaglyphenstereo grundsätzlich implementiert wird. Anschließend werden wir auf die Berechnung der Stereoansichten ausgehend von einer einzelnen, vorgegebenen Ansicht eingehen, was wiederum ohne Änderung des Quelltexts der Anwendung erfolgen wird. Da die Szene zweimal gezeichnet werden muss, werden wir dann Alternativen diskutieren, um ein Neuzeichnen der Szene zu erzwingen. Eine ereignisbasierte Lösung wird im Detail betrachtet werden. Eine Beurteilung des Ansatzes wird zeigen, dass die Lösung nahezu optimale Leistung erzielt. Die Diskussion endet mit einer Beschreibung von Möglichkeiten, um die Stereoparameter interaktiv anzupassen.

Kapitel 4: Gitterkonvertierung

In diesem Kapitel beschreiben wir Techniken zur Konvertierung von Gittern in ein Format, das die Behandlung durch effizientere Algorithmen ermöglicht. Wir werden argumentieren, dass durch den hierdurch erzielten Geschwindigkeitszuwachs unter bestimmten Bedingungen Fernvisualisierung vermieden werden kann, und dass Hierarchien von kartesischen Gitter ein gutes Gleichgewicht zwischen Genauigkeitsverlust einerseits und erhöhter Leistung andererseits darstellen. Wir werden dann eine detaillierte Beschreibung eines Werkzeugs zur Gitterkonvertierung geben, das für diese Arbeit im Hinblick auf Anwendung in der Automobilindustrie entwickelt wurde.

Wir beginnen die Diskussion mit einem Kurzüberblick über Softwarepakete zur Strömungssimulation und die resultierenden Gittertypen. Die Erläuterungen werden die Wahl von Hierarchien kartesischer Gitter weiter motivieren. Die Diskussion wird weiterhin zeigen, dass bereits effiziente Algorithmen für diese Art von Gittern existieren und dass die Konvertierung von Gittern den Weg für eine interaktive Visualisierung vieler Datensätze ebnet, die ansonsten nur unbefriedigend analysiert werden könnten.

Wir werden dann bestehende Konvertierungswerkzeuge betrachten und deren Vor- und Nachteile untersuchen, bevor wir mit der Beschreibung zweier eigener Verfahren fortfahren: der halbautomatischen Gitterkonvertierung und der vollautomatischen, adaptiven Gitterkonvertierung.

Halbautomatische Konvertierung setzt die Definition von Verfeinerungsregionen durch den Benutzer voraus. Eine Beurteilung wird zeigen, dass dieser objektbasierte Ansatz schnell und einfach zu implementieren ist, dass er jedoch auch ineffizient bzgl. der Anzahl der generierten Zellen und der erzielten Genauigkeit ist. Die Notwendigkeit für einen adaptiven Ansatz wird hierdurch deutlich werden.

Wir werden dann argumentieren, dass ein geometriebasierter Ansatz im Hinblick auf Geschwindigkeit gegenüber einem fehlerbasierten Ansatz zu bevorzugen ist, und wir werden drei wesentliche Phasen als Teil der Konvertierung identifizieren: den Aufbau eines Oktalbaumes, die Korrektur des Oktalbaumes und die Dateninterpolation. Für den Aufbau des Oktalbaumes werden wir gängige Oktalbäume aufgrund des hohen Speicherbedarfs als ungeeignet klassifizieren was uns dazu führen wird, Notationen und Definitionen für einen effizienten zeigerlosen Oktalbaum zu entwickeln. Wir werden im Folgenden dann auf einige Standardoperationen und deren Implementierung für zeigerlose Oktalbäume eingehen. Die Notwendigkeit eines Korrekturschritts werden wir damit begründen, dass bestehende Visualisierungswerkzeuge beliebige Oktalbaumstrukturen nicht artefaktfrei darstellen können, und wir werden Algorithmen vorstellen, die die effiziente Korrektur zeigerloser Oktalbäume erlauben. Schließlich werden wir auf die Interpolationsphase eingehen und eine parallele Implementierung des adaptiven Konvertierungsansatzes beschreiben.

Das Kapitel endet mit einer vergleichenden Beurteilung beider implementierter Ansätze im Hinblick auf Laufzeit und die erzielte Genauigkeit im Verhältnis zur resultierenden Zellanzahl.

Kapitel 5: Merkmalsextraktion

Dieses Kapitel befasst sich mit der Extraktion von Strömungsmerkmalen. Wir werden argumentieren, dass die Visualisierung von Strömungsmerkmalen aufgrund der einhergehenden Datenreduktion und leicht zu interpretierender Ergebnisse vorteilhaft ist. Die Diskussion wird jedoch auch zeigen, dass die betreffenden Algorithmen rechenintensiv sind.

Die Diskussion wird beginnen mit einer Beurteilung von Strömungssimulationsverfahren im Hinblick auf Rauschen, und sie wird zeigen, dass sowohl experimentelle als auch numerische Methoden zur Generierung von Strömungsdatensätzen anfällig für Rauschen sind. Wir werden in diesem Zusammenhang Methoden wie LDA, PIV, DNS, LES und RANS begutachten. Die gewonnenen Er-

kenntnisse werden wir nutzen, um die Notwendigkeit für Datenvorfilterung und Datenentrauschen und die Möglichkeit der schnellen Merkmalsextraktion – und insbesondere Wirbelextraktion – zu motivieren.

Wir fahren dann fort mit einer Beschreibung einer optimierten Softwareimplementierung des λ_2 Verfahrens. Zu diesem Zwecke werden wir genau auf die effiziente Berechnung des Geschwindigkeitsgradiententensors und die effiziente Berechnung von Eigenwerten eingehen. Eine Beurteilung der Implementierung wird zeigen, dass trotz der Optimierungen eine interaktive Merkmalsextraktion für typische Datensatzgrößen der Praxis unmöglich ist.

Wir werden daher im Folgenden eine GPU-basierte Implementierung des λ_2 Kriteriums vorstellen, die sowohl das Entrauschen und die Wirbelerkennung als auch die Visualisierung umfasst. Die Auswahl geeigneter Filter und die Implementierung separierbarer Filter wird dann genauer diskutiert werden bevor wir Strategien zur Portierung der Eigenwertberechnung auf die GPU vorschlagen. Für die Visualisierung werden wir verschiedene Visualisierungsmethoden beurteilen und schließlich eine Volumenvisualisierungslösung entwickeln, die einen guten Kompromiss zwischen Geschwindigkeit und Qualität darstellt. Die Lösung wird ohne Eingriffe durch die CPU auskommen. Die Implementierung wird im Hinblick auf die Softwarelösung bewertet werden, was zeigen wird, dass durch den Einsatz der Graphikhardware interaktives Arbeiten möglich ist.

Da Wirbel oft mittels Isoflächen dargestellt werden, werden wir uns im Folgenden GPU-beschleunigten Techniken zur Extraktion von Isoflächen widmen. Hierzu werden wir zunächst eine GPU-basierte Implementierung des *Marching Tetrahedra* Algorithmus zur Extraktion von Isoflächengeometrie entwickeln und dann ein ebenfalls GPU-basiertes Verfahren zur Strahlverfolgung in Volumen erläutern. Es wird sich zeigen, dass die letztgenannte Implementierung nicht auf die Visualisierung von Isoflächen beschränkt ist. Eine Reihe von Beispielen zur sowohl direkten als auch indirekten Volumenvisualisierung wird dies bestätigen. Beide Verfahren werden kritisch beurteilt werden.

Das Kapitel endet mit einem neuen Verfahren zur Segmentierung von Wirbelstrukturen. Mit diesem Verfahren ist es möglich, eine Filterung auf Wirbelbasis durchzuführen was sowohl die Visualisierung an sich beschleunigt als auch die Datenanalyse, da es dem Forscher ermöglicht wird, sich auf relevante Strukturen zu konzentrieren.

Wir werden die Diskussion beginnen mit einer Übersicht über verwandte Arbeiten und hierdurch den Bedarf für verbesserte Wirbelsegmentierung motivieren. Die Kombination von *Predictor-Corrector* Verfahren mit der λ_2 Methode wird dann als Mittel zur verlässlichen Erkennung und Segmentierung von Wirbeln identifiziert werden. Auf die Implementierung wird im Folgenden detailliert eingegangen werden, gefolgt von einer Beurteilung mit Hilfe von Daten der Strömungsforschung.

Im weiteren Verlauf werden wir die Notwendigkeit für verbesserte Interaktionsmöglichkeiten bzgl. der segmentierten Wirbelstrukturen begründen und den Entwurf und die Implementierung eines Analysewerkzeuges beschreiben, das die Auswahl, das Ausblenden, die Filterung und Manipulation von Wirbelstrukturen ermöglicht. Das System wird vervollständigt durch gängige Verfahren der numerischen Strömungsvisualisierung, um den Kontext, d.h. die einbettende Strömung, darzustellen. Hierzu werden Techniken wie Schnittebenen, Pfeildarstellungen der Geschwindigkeitsvektoren, LIC, Farbkodierungen, Teilchenbahnen als auch Isoflächen verwendet werden. Zur Unterstützung der Erkennung komplexer, verflochtener Strukturen wird außerdem Stereovisualisierung eingesetzt werden.

Die Darstellung endet mit einer Beurteilung des System durch Anwender der Strömungsforschung.

Kapitel 6: Fazit

Das letzte Kapitel fasst die für diese Arbeit entwickelten Algorithmen und Techniken zusammen. Um die Auswahl geeigneter Beschleunigungstechniken für den betreffenden Fall zu erleichtern, werden wir das Fazit in Form eines Flussdiagramms geben, das – ausgehend von einer Reihe von Entscheidungen in Abhängigkeit individueller Rahmenbedingungen – eine Empfehlung liefert.

Die Arbeit schließt mit einem Ausblick und Empfehlungen für zukünftige Forschungsarbeiten.

Chapter 1

Introduction

Flows play an important role in many areas of science and technology. In the car manufacturing industry mass flows are studied for determining the degree to which engines are cooled; in chemical engineering mixing processes of fluids are studied to optimize the synthesis of new substances; and in medicine, blood flows are analyzed for examining the effects of vessel narrowing. In either case, independent of whether the application goal is designing devices or optimizing processes, a thorough understanding is required of the flow and the structures interacting therein. This understanding can be gained by theoretical discussions, a research field commonly referred to as fluid dynamics, which in combination with high-performance supercomputers and multi-PC cluster computers allows for highly accurate numerical flow simulations. Alternatively, insight can be acquired by physical flow visualization, the process of measuring and imaging physical flows. While when using the latter in a qualitative way, easy to comprehend images are often readily available, for the former some form of numerical flow visualization turning mere collections of numbers into more meaningful visual representations is inevitable. And so it is for quantitative physical flow visualization.

Increasing processing power of supercomputers and cluster computers built of off-the-shelf hardware as well as improved measurement techniques, however, have led to data set sizes with which conventional numerical flow visualization can no longer keep up. Numerical flow visualization in this respect is no exception and similar problems caused by large data sets occur in other research fields of computer graphics and in general computer science as well, ranging from photo-realistic rendering of detailed building models over the administration and querying of large data bases in data base design and the understanding of software comprising millions of lines of code in software engineering. In either case, what "large" actually means strongly depends on the application context and, in particular, the data analysis task. Engineers studying the flow around car bodies require

quantitative information in order to make reliable estimations about fuel consumption; thus, highly resolved data sets are examined often comprising tens or even hundreds of millions of data points. On the other hand, fluid dynamics researchers studying flow transitions and turbulence in the first instance strive for a qualitative understanding of the formation and interaction of structures. For these purposes, computationally obtained data sets of a few million cells are sufficient. And researchers measuring actual physical flows have to content themselves with data sets of a few hundred thousand to even a few thousand data points. Thus, when working with flows data size differences of two or three orders of magnitude are common in practice.

The term “large”, however, must also be judged in the context of available hardware resources and the complexity of the algorithms that have to be applied to the data to attain the desired visualization. So, for example, a data set comprising tens of millions of cells will generally be considered large when standard personal computers have to be used for visualization while when using a supercomputer data of this size is not intimidating. Similarly, extracting vortices from a turbulent data set of several thousand data points can be done instantaneously while detecting the structures in data a hundred times larger will consume a considerable amount of time. Therefore, whether there is a need for acceleration techniques is dependent on a number of parameters like complexity of operations and data set size. In general, however, the need for improvements manifests itself as soon as interactivity is lost or—in a broader sense—if reaching the final goal of understanding the flow and improving device designs or processes becomes increasingly difficult and less effective. The opening gap between, on the one hand, ever increasing data set sizes and, accordingly, ever increasing information presented to the user and, on the other hand, limited human cognition capabilities therefore presents another issue which acceleration techniques for numerical flow visualization must address.

The different scenarios clearly illustrate the need to tackle the problem of accelerating numerical flow visualization on a number a different levels utilizing highly diverse approaches. And it is this line that is adopted in this work. Regarding the largest data sets, we present novel solutions for accomplishing the visualization and data analysis with the help of remote hardware. Alternatively, feature-based visualization is provided capable of presenting the relevant information contained in the data in a highly condensed way. For medium size data, in turn, parallel out-of-core algorithms tailored to industrial needs are presented for modifying data set structures, thereby opening the way to more efficient algorithms at the cost of decreased accuracy. Finally, for medium size to small size data we present algorithms and implementations making highly efficient use of locally available hardware.

As will be shown, the followed multi-level approach enables researchers to

effectively use the hardware resources at hand and to freely choose the level of interference with respect to both data, visualization techniques, and visualization applications—ranging from none at all to visualizations of a highly compact representations derived from the original data.

1.1 Thesis Overview

In Chapter 2 we provide the necessary background information on technologies and techniques utilized in this work. This includes a discussion of the visualization pipeline transforming raw data into images, techniques for visualizing vectorial flow data and scalar volume data, and related software and hardware architectures involved in this process.

In Chapter 3, we propose two generic remote visualization solutions for taking advantage of remotely located graphics resources. Each solution proves beneficial in certain application scenarios. The description is complemented by a discussion of useful add-ons which—again without violating genericity—allow for integrating stereo capabilities and to non-intrusively adjust user interfaces to the target display device. These solutions mark the lowest level of acceleration techniques and keep the data at the producer, e.g., a supercomputing center. Remote visualization is a fall-back for either extremely large data sets that can only be handled by dedicated machines or for arbitrarily sized data sets that need to be visualized on computers or display devices with very low processing power or hardware resources.

Chapter 4 discusses techniques on the next higher level which proceed by transforming the raw data into a format allowing for more efficient visualization, the idea being that although the local hardware resources might be insufficient for visualizing the original data, interactive visualizations of the transformed data will become possible. Accordingly, on this level the flow data is transferred to the user. Since the data conversion introduces errors, this approach trades speed for accuracy.

While on the previous levels the velocity field is visualized, the acceleration techniques discussed in Chapter 5 visualize flow features or, more precisely, vortices. The benefit of visualizing features is an inherent data reduction allowing for more efficient visualization exclusively presenting relevant information. The discussion will elaborate on how these flow features can be efficiently computed and how the resulting data can be efficiently visualized. The chapter concludes with a discussion of techniques for separating and interacting with individual flow features which allows for a further information reduction.

The work closes in Chapter 6 with a set of recommendations giving hints about when and how to use and combine the presented approaches.

1.2 Acknowledgments

Good research is often the product of many interested people brought together. And it was in this case. I am most grateful to my advisor, Thomas Ertl, who once told me that a PhD thesis cannot be pre-loaded, thereby returning me to the right track; to my co-examiner Frits Post (TU Delft) for extensive proof-reading and for making a one-day round trip to Germany despite tropical temperatures; and to all my coauthors, namely (in alphabetical order): Ralf Botchen (HipHop? That is a kind of Techno, isn't it?), Min Chen (Swansea University), Joachim Diepstraten ("Only one person answered for the original version yes often and all the others said seldom or never in contrast for our replacement menue [sic] three persons said yes often or always and only one answered sometimes." Remember?), David S. Ebert (Purdue University), Mike "The Nexus Guy" Eissele, Kelly P. Gaither (University of Texas), Jingshu Huang (Purdue University), Yun "Wow, a Benz!" Jang (Purdue University), Thomas "MFG" Klein (world-class system administrator and just as brilliant flow topology researcher), Marcelo Magallón (let's just say `awk` and `perl` are equally useful), Guido Reina (Barco AG?), Ulrich Rist (IAG, University of Stuttgart), Dirc "Together again?" Rose, Martin Schulz (science+computing AG), Magnus "The Toe" Strengert ("We cannot descend here, it's way too steep."—"We can, I'll show you."—"Don't do it!"—"It's easy ... see? ... aaahh!"), Manfred Weiler (wrote the first GPU-based program justifying the unit frames/hour and most of the time busy with class reunions), and Daniel "Danny" Weiskopf (texture advection god and also my favorite beach volleyball team mate).

I am also grateful to my long-term office mate Martin Rotard for bearing with me, to my short-term office mate Tobias Schafhitzel for introducing me to climbing, and to Martin Kraus who returned just in time to have a last glance at the exposition and the layout (and it still amazes me how anyone can be that picky!). Finally, doing research would not have been fun without a nice team of colleagues making for a pleasant atmosphere. Thank you all!

Chapter 2

Graphics and Visualization Techniques

This section is supposed to give an overview of the (flow) visualization problem, the kind of data that was processed during this work, and various visualization-related techniques and technologies that have been employed for the visualization. The section serves a double purpose: First, to illustrate the complexity of the operations involved in the process of generating the final image and, thus, to motivate the need for improvements and acceleration techniques and, second, to provide a compact reference on visualization-related topics for forthcoming chapters.

2.1 The Visualization Pipeline

Visualization is the process of making visible what is otherwise invisible or hard to extract from given abstract data and information. Its use is ubiquitous but often overseen. A nice example are weather maps shown in news programmes and newspapers. In principle, we could look directly at the photographs shot by satellites. However, this data would hardly reveal anything more than which areas have cloudy skies and which ones have clear skies. Of course, we could also consult infrared images to find out about current temperatures. While this would improve our understanding of the current weather conditions, it surely would not be comfortable to consult several data sources and it surely would not tell us about tomorrow's weather since the information is not presented in a format easily understood by the layman. Visualization alleviates both of these problems. Regarding the former, visualization transforms data (rainfall, wind speed, temperature) collected by satellites and meteorological stations to meaningful graphical illustrations understood by a public audience (Figure 2.1). In addition, regarding the latter, visualization also includes the assembling of data from various sources since this also

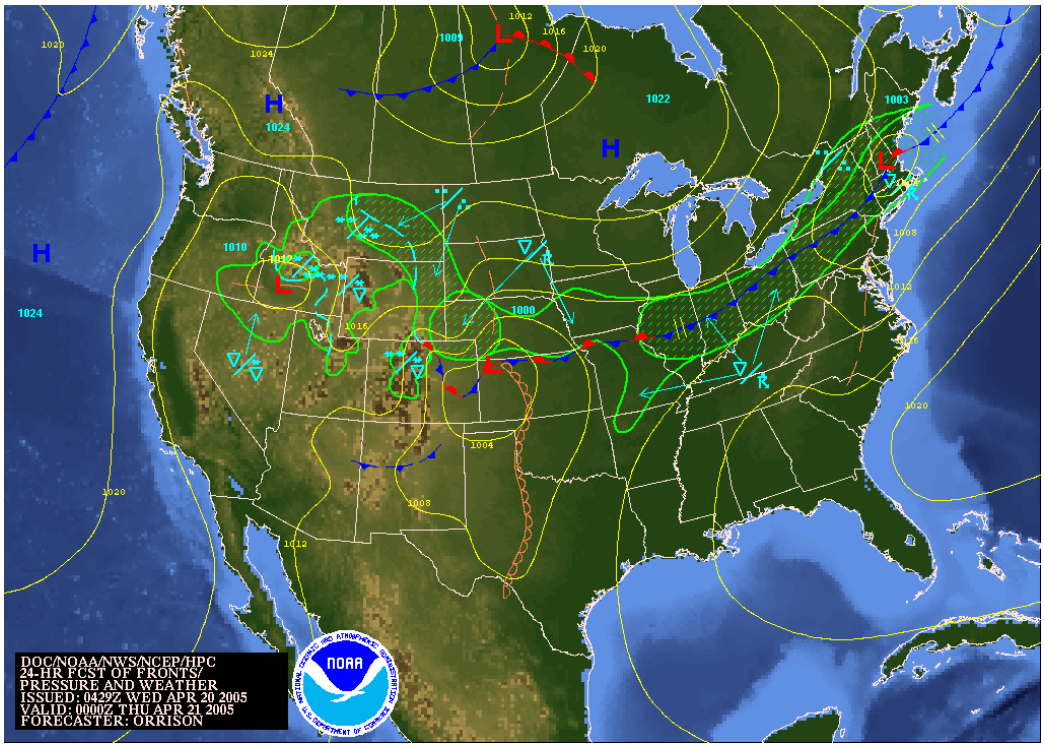

Figure 2.1 Various visualization techniques of highly diverse data combined into a single image. Image courtesy National Atmospheric and Oceanographic Administration (NOAA).

improves the data understanding.

The example illustrates that data can be obtained from several sources. The data used for creating weather maps will include both information from measurements (current air temperatures, wind speeds) and data from simulations/computer models using real-world data to project the current weather conditions into the future to make weather forecasts. Obviously, the kind of data also differs: isobars visualize scalar pressure data while wind is described with multi-component velocity data informing about both wind speed and direction. The visualization must thus accommodate to different requirements and provide a set of visualization techniques from which the optimal technique is to be chosen.

Besides the mentioned data sources, visualization techniques can and are also applied to data obtained from data bases rather than simulation or measurements, e.g., historical stock charts in economics. We will not consider these data sources but rather, in this work, would like to understand scientific visualization as the process of making visible by means of computer-generated images information included in data provided by the natural and engineering sciences. Thus, we ad-

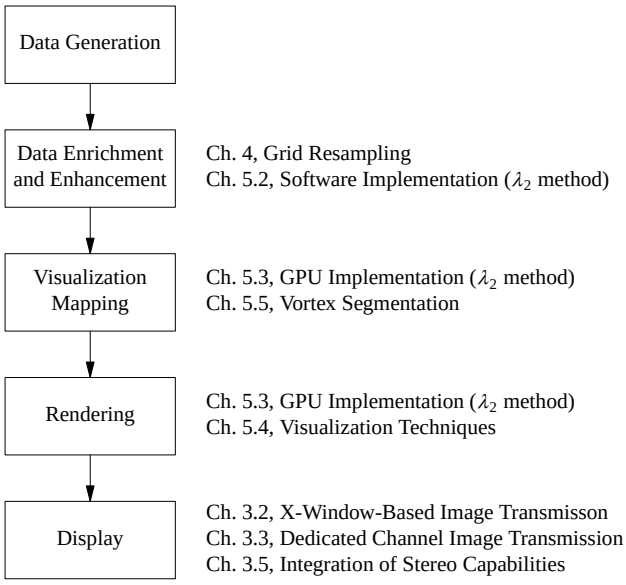

Figure 2.2 Various stages scientific data undergo during visualization on their way from the data source to the final image. The chapter numbers to the right show which stage is tackled in which part of the work.

dress meteorological data as seen above as well as sensor data from medicine (CT, MRI, ultra sound data) and simulation data from physics like cosmological research about the formation of stars.

Independent of the data and their source, the data usually undergo a series of transformations on their way to the final image, a process often depicted by the so-called visualization pipeline [28] shown in Figure 2.2 and elaborated on in the following. The acceleration techniques developed in this work have been devised in close consideration of this pipeline in order to tackle the problem on several levels. The figure gives the chapter numbers and subchapter titles where the descriptions of the respective work packages can be found.

Depending on the data source the visualization of the raw data will require some kind of filtering or pre-processing to reduce the amount of data or to improve the information content. Typical tasks include denoising, conversion, and segmentation. In the example of weather data, some meteorological station might operate erroneous equipment generating faulty data; or satellite data of water depths might exhibit random inaccuracies due to various environmental influences. The first kind of data should not find its way into the visualization at all and must be extracted in a pre-processing step. In the second case filtering might be used to remove the oscillations.

The next step is termed *visualization mapping*. It includes two mapping aspects. Conceptual mapping answers questions about what is to be seen and about

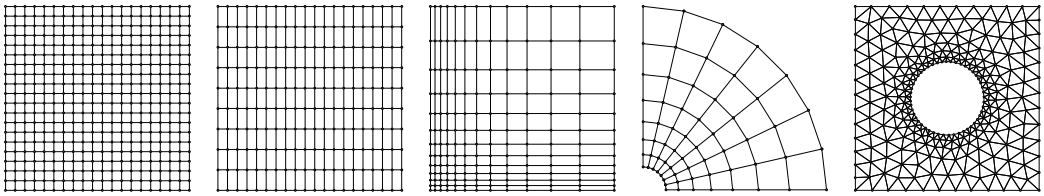

Figure 2.3 Common grid types found in scientific visualization. From left to right: Cartesian grid; regular or uniform grid; rectilinear grid; structured or curvilinear grid; unstructured grid. Arbitrary combinations are possible. Unstructured grid data courtesy John Burkardt, Florida State University.

how it should be visualized. The actual mapping includes the computation of derived quantities suitable for visualization and the mapping of this data to graphical primitives. In the example, we might express the desire to visualize wind velocities in a certain area by arrows whose directions match the wind directions and whose lengths and colors indicate the wind speeds. Thus, we interpolate the required number of wind velocity vectors based on the available information and generate the required line segments depicting the arrow heads and tails. Since velocity magnitudes do not canonically map to a certain set of colors, we would also have to derive a suitable color scheme as part of the visualization mapping stage.

Once the set of graphics primitives (points, lines, triangles) has been compiled, the primitives are mapped into displayable images. This step involves passing the generated vertices—points in space—through another pipeline, the so-called graphics pipeline where the vertices undergo viewing transformations and lighting calculations, amongst others. We will elaborate on this pipeline in Section 2.5.

The final step is the display of the rendered images. This is usually accomplished by direct output of the rendering process but it also involves playback of pre-computed animation sequences in the form of videos.

As can be seen in Figure 2.2, the acceleration techniques proposed in this work address all levels except for the data generation stage. This is understandably so since performing accurate measurements and deriving theoretical models describing natural phenomena as close as possible is the task of physics, geo-sciences, and medicine rather than that of computer science.

2.2 Grid Types and Interpolation

Data—independent of their source—are usually provided as a set of tuples consisting of a spatial position and one or more data values of arbitrary type. If we

draw imaginary lines from a data point to its closest neighbor(s)[1], a *grid* is obtained that can be classified according to its topology and the shape of its grid cells. This section gives an overview of selected grid types and elaborates on interpolation schemes for determining new data values within grid cells.

2.2.1 Grid Types

The grid types relevant for this work are depicted in Figure 2.3. We will present these types in increasing order of generality [89].

The most simple grid type is the Cartesian grid. In this grid type, the distance between grid points has unit length in all directions and lines through grid points are either parallel or perpendicular to each other. That is, the individual cells of a 3D Cartesian grid are cubes. World coordinates for this grid type are given by (i, j, k). The absence of scaling factors indicates that Cartesian grids are often not intended to be mapped to (physical) world coordinates.

Cartesian grids are the preferred grid type in many applications (including scientific visualization) since they can significantly simplify algorithms. First, storing the grid can be accomplished with a simple 3D array. Accordingly, accessing the data is also efficient. And second, derivatives can be easily and efficiently approximated by forward/backward and central differences at the boundaries and the interior with errors of the order of $\mathcal{O}(h)$ and $\mathcal{O}(h^2)$, respectively, as is easily shown with a Taylor expansion:

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \mathcal{O}(h^2) \,, \\ f'(x) &= \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \,. \end{aligned} \tag{2.1}$$

And analogously for backward differences:

$$\begin{aligned} f(x-h) &= f(x) - hf'(x) + \mathcal{O}(h^2) \,, \\ f'(x) &= \frac{f(x) - f(x-h))}{h} + \mathcal{O}(h) \,. \end{aligned} \tag{2.2}$$

The error when computing derivatives with one-sided differences is $\mathcal{O}(h)$ although the function values have been evaluated with an error of $\mathcal{O}(h^2)$ due to the sample distance h being in the denominator. However, if function values are available both to the right and to the left, a more accurate approximation of the order of $\mathcal{O}(h^2)$ can be attained with central differences:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \mathcal{O}(h^3) \,,$$

[1] In the following when talking about distances between grid points we will always mean distances along these imaginary lines.

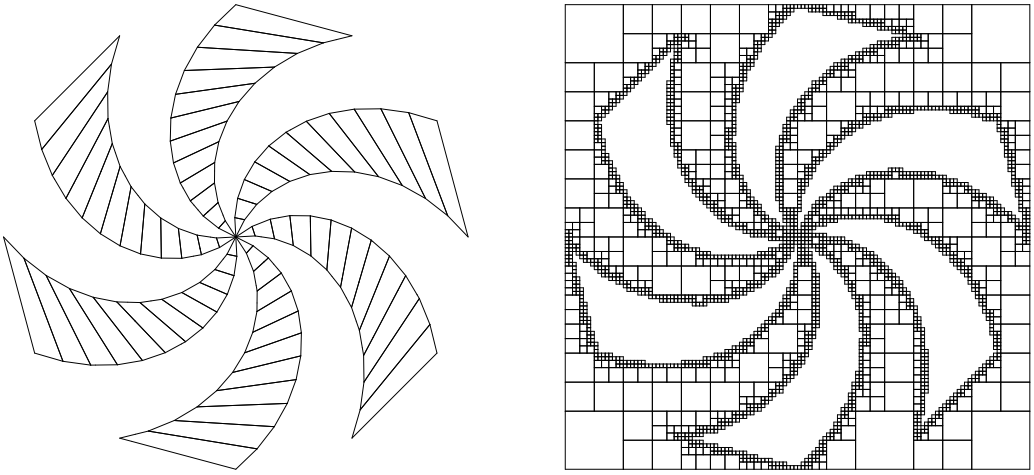

Figure 2.4 Left: Turbine-like geometry made up of non-rectangular, non-axis-aligned quadrilaterals. Right: The geometry resampled to a grid of rectangular, axis-aligned grid cells on a square domain.

$$\begin{aligned} f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2!}f''(x) + \mathcal{O}(h^3) \,, \\ f'(x) &= \frac{f(x+h) - f(x-h))}{2h} + \mathcal{O}(h^2) \,. \end{aligned} \tag{2.3}$$

Uniform or regular grids are similar to Cartesian grids but the grid has been scaled non-uniformly, i.e. the grid point distance in one dimension differs from the distance in at least one other dimension and, accordingly, grid cells may be cuboids. Again, the grid point coordinates and the derivatives can be easily computed. World coordinates are given by $(i \cdot \Delta x, j \cdot \Delta y, k \cdot \Delta z)$ where Δx, Δy, and Δz denote the grid point distances in the respective dimension. If the grid point with indices $(0, 0, 0)$ does *not* coincide with the origin, the final world coordinates are obtained by additionally applying a constant translation $(x_0, y_0, z_0)^T$. Regular grids are used throughout Chapter 5.

While Cartesian and uniform grids exhibit constant grid point distances in each dimension, in rectilinear grids the grid point distance may be arbitrary. Thus, while storing the grid is still simple, computing world coordinates requires look-ups into coordinate arrays: $(x[i], y[j], z[k])$. Rectilinear grids are advantageous in, e.g., the simulation of the flow in the boundary of a flat plate where the resolution must be high near the plate to resolve fine flow structures and where a coarse grid suffices in outer regions where the flow becomes increasingly laminar.

Rectangular grids are often unfavorable if the data volume is bounded by some geometry as it is obtained when measuring or simulating the flow inside a bent tube. As shown in Figure 2.4, approximating any geometry not parallel to the

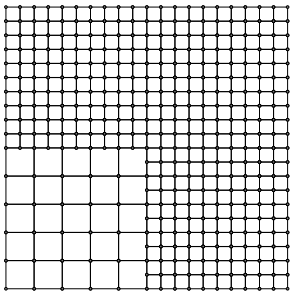

Figure 2.5 Hierarchical Cartesian grid. Note that the shown grid is *not* a combination of different grid types but rather a combination of different resolutions.

grid axes results in a vast number of grid points. In fact, an exact representation even requires infinitely many grid points. The problem can be solved by allowing for non-rectangular grid cells without having to sacrifice the advantage of simple storage of less general rectangular grids. The grid type is called structured or curvilinear. However, since now grid points with the same index in an arbitrary dimension do not necessarily have the same coordinate value in this dimension, computing world coordinates depends on all indices for every coordinate: $(x[i,j,k], y[i,j,k], z[i,j,k])$. Structured grids are widely found in the aerospace and car manufacturing industry.

The most general grid type does not impose any restrictions on the shape of the grid cells. In practice, however, these so-called unstructured grids usually only comprise tetrahedra, hexahedra, prisms, and pyramids. Since unstructured grids have an irregular topology, no connectivity information is given implicitly and storing the grid becomes less efficient. As for rectilinear grids, computing world coordinates requires lookups into coordinate arrays: $(x[i], y[i], z[i])$.

It is often more efficient—both with regard to memory usage and processing time—to combine any of the above grid types. These grids are called *hybrid*. If two grids of the same type but different resolution are combined to form a new grid, we speak of a *hierarchical* grid (Figure 2.5). We will have more to say about hybrid and hierarchical grids in Section 4.

In the above discussion, it was assumed that data values are valid only at the corresponding grid point. In another view, data points define the center of a grid cell and the data value is assumed to be valid everywhere within the enclosing cell. Therefore, in this *cell-centered* view the dual grid is used of what is obtained when the data is regarded *vertex-based*. Cell-centered data is usually disadvantageous since data values change abruptly when crossing grid cell borders. Cell-centered data is thus often converted to vertex-based data by interpolating grid point values from values defined in neighboring cells (see next section). While this approach often leads to more pleasing visualizations, it neither improves the data content

nor the grid resolution and might thus be misleading.

2.2.2 Grid Interpolation

Many visualization algorithms require data values to be known not only at grid points but also at arbitrary locations inside grid cells. Estimating these data values is the task of interpolation methods. In general, interpolation methods proceed by fitting a function through the data points from which the new value is to be interpolated and determine the sought-after value by evaluating the function at the interpolation position. Obviously, the more points are used for determining the new value, the better the approximation will be.

In *nearest-neighbor interpolation* the data point closest to the interpolation position is used to interpolate the new data value. Since only a single data point is used, the new data value matches the value of this data point.

Linear interpolation uses two data points and fits a linear function to these points. Given two points $\mathbf{x}_a$ and $\mathbf{x}_b$ with respective data values $s(\mathbf{x}_a) = s_a$ and $s(\mathbf{x}_b) = s_b$, the value $s(\mathbf{x})$ at a position $\mathbf{x}$ somewhere between $\mathbf{x}_a$ and $\mathbf{x}_b$ is interpolated by:

$$s(\mathbf{x}) = \frac{|\mathbf{x}_b - \mathbf{x}|}{|\mathbf{x}_b - \mathbf{x}_a|} s_a + \frac{|\mathbf{x} - \mathbf{x}_a|}{|\mathbf{x}_b - \mathbf{x}_a|} s_b \; . \tag{2.4}$$

Linear interpolation can be easily extended to 3D for interpolation in cubical grid cells. Considering again Equation 2.4, it is seen that, first, the interpolation position splits the line connecting the two data points into two smaller line segments and, second, that the weight of a data point is equal to the ratio of the length of the opposite line segment and the length of the line segment connecting the two data points. In 3D, if we draw three lines through the interpolation position parallel to the grid cell axes, the grid cell virtually is divided into eight subcells. Analogously to the 1D case, the weight of a data point now is the ratio of the volume of opposite subcell and the grid cell volume and the interpolated data value is obtained by summing over the weighted data point values. Thus, for a grid cell of the dimensions $\Delta x \times \Delta y \times \Delta z$ of a rectilinear grid and an interpolation position given by $\mathbf{x} = (\alpha, \beta, \gamma)$ the interpolated value $s(\mathbf{x})$ is given by

$$\begin{aligned}
s(\mathbf{x}) = (1-\alpha')(1-\beta')(1-\gamma') \quad & s_{000} \quad + \\
(1-\alpha')(1-\beta')\gamma' \quad & s_{001} \quad + \\
(1-\alpha')\beta'(1-\gamma') \quad & s_{010} \quad + \\
(1-\alpha')\beta'\gamma' \quad & s_{011} \quad + \\
\alpha'(1-\beta')(1-\gamma') \quad & s_{100} \quad + \\
\alpha'(1-\beta')\gamma' \quad & s_{101} \quad +
\end{aligned}$$

$$\begin{array}{rl} \alpha'\beta'(1-\gamma') & s_{110} \quad + \\ \alpha'\beta'\gamma' & s_{111} \end{array} \tag{2.5}$$

with $\alpha' = \text{frac}(\alpha/\Delta x)$, $\beta' = \text{frac}(\beta/\Delta y)$, and $\gamma' = \text{frac}(\gamma/\Delta z)$, where $\text{frac}(x)$ denotes the fractional part of x. For the grid point values we used the notation s_{xyz} which is meant to denote the value of the grid point whose index is $(i+x, j+y, k+z)$ under the assumption that the grid point with the smallest indices in any dimension has the index (i, j, k). This interpolation method is called *trilinear interpolation*. Trilinear interpolation is used in Chapter 5.

Although trilinear interpolation often provides sufficient accuracy, some applications require higher-order interpolation. *Lagrange interpolation* can be used to fit a polynomial P of order $n-1$ to any n points $y_1 = f(x_1), \ldots, y_n = f(x_n)$:

$$\begin{aligned} P(x) &= \sum_{j=1}^{n} P_j(x) \ , \\ P_j(x) &= y_j \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{x - x_k}{x_j - x_k} \ . \end{aligned}$$

As with linear interpolation, the method can be extended to 3D for rectilinear grids by successively applying the interpolation to higher dimensions until the interpolation position has been reached. This scheme is used in Section 5.5 with a four-point Lagrange interpolation.

The above methods cannot be applied to grids with non-rectangular cells. However, the basic idea of computing the weighting factor based on the ratio of subvolumes and the cell volume still can be used. For the moment we will assume that we are given a 3D unstructured grid with tetrahedral cells. The four grid points are denoted by $\mathbf{x}_a$, $\mathbf{x}_b$, $\mathbf{x}_c$, and $\mathbf{x}_d$, the tetrahedron enclosing the interpolation position $\mathbf{x}$ is denoted by $\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d$. We can obtain a subdivision of the tetrahedron by drawing lines from $\mathbf{x}$ to all the grid points comprising the tetrahedron. These subvolumes again are related to the total volume of the tetrahedron $\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d$ to obtain the individual weights. Let $|\cdot|$ be an operator that, when applied to a tetrahedron, returns its volume. The interpolated value $s(\mathbf{x})$ is then given by:

$$s(\mathbf{x}) = \frac{|\mathbf{x}\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d|}{|\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d|} s_a + \frac{|\mathbf{x}\mathbf{x}_a\mathbf{x}_c\mathbf{x}_d|}{|\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d|} s_b + \frac{|\mathbf{x}\mathbf{x}_a\mathbf{x}_b\mathbf{x}_d|}{|\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d|} s_c + \frac{|\mathbf{x}\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c|}{|\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d|} s_d \ . \tag{2.6}$$

The interpolation weights are called the *barycentric coordinates* of $\mathbf{x}$ with respect to the tetrahedron $\mathbf{x}_a\mathbf{x}_b\mathbf{x}_c\mathbf{x}_d$ and they obviously sum up to one. Although Equation 2.6 only applies to tetrahedra, this kind of interpolation is of special importance since any polyhedron can be decomposed into tetrahedra, any of which will

Figure 2.6 Physical flow visualization: Photograph of a liquid flow through a circular tube with segmental baffles visualized with fluorescent dye [3].

contain the interpolation position; thus, interpolation with barycentric coordinates can be used to interpolate in arbitrary grids.

If we want to avoid the overhead involved in subdividing polyhedra into tetrahedra then *inverse distance weighting* can be resorted to. The idea of this interpolation method is to consider the distance to the data points used for the interpolation and to assume that the closer a data point is to the interpolation position, the more important it should be for interpolating the new value. Thus, if the n data points and corresponding weights are given by $\mathbf{x}_i$ and s_i, $i \in \{1, \ldots, n\}$, respectively, then the interpolated value $s(\mathbf{x})$ is given by

$$\begin{aligned} s(\mathbf{x}) &= \sum_{i=1}^{n} \frac{s_i}{D \cdot d_i}\,, \\ D &= \sum_{j=1}^{n} \frac{1}{d_j}\,, \end{aligned} \tag{2.7}$$

where d_i denotes the (Euclidean) distance between $\mathbf{x}$ and $\mathbf{x}_i$. If any $d_i = 0$ the value $s(\mathbf{x})$ as defined by Equation 2.7 is undefined. However, $d_i = 0$ means that the interpolation position coincides with a polyhedron vertex; thus, in this case $s(\mathbf{x}) = s_i$.

Both interpolation based on barycentric coordinates and inverse distance weighting can be employed for solving the above problem of converting cell-centered data to vertex-based data (Section 2.2.1). In Chapter 4 the latter approach is used.

2.3 Overview of Flow Visualization

This work concentrates on the visualization of flows. Flows are found in most natural sciences and at highly different scales: in medical applications blood flows through veins and vessels are examined, in geo-sciences ocean currents represent the flow. In either case, the flow is described by a volumetric data set, a field, where each grid point is assigned a 3D component velocity vector indicating the

direction of the flow and its speed:

$$\boldsymbol{v}(\boldsymbol{x},t) = \begin{pmatrix} u_1(\boldsymbol{x},t) \\ u_2(\boldsymbol{x},t) \\ u_3(\boldsymbol{x},t) \end{pmatrix} .$$

The field may be given analytically in rare cases but usually it is given in discrete form, representing the corresponding continuous field by a large number of samples. In the latter case, the samples can be obtained by either experiment—a process called *physical flow visualization* or *experimental flow visualization*—or by computer simulations. The process of visualizing flow data by means of computer-generated images is called *computer-aided flow visualization* or *numerical flow visualization*. Flow data is often accompanied by a number of scalar fields of equal resolution describing other physical properties of the moving gas or fluid like density, temperature, or pressure. The following sections give an overview of techniques for visualizing flows.

2.3.1 Physical Flow Visualization

Physical flow visualization techniques [69] can be divided into two classes: intrusive techniques and non-intrusive techniques. We will define intrusive techniques as techniques where the fluid flow (its density, temperature, etc.) might be disturbed either by adding material or by adding energy in an amount exceeding that of visible light required for the observation[2]. Examples for intrusive techniques are:

- *Streak Lines*. In this technique, dye is continuously released into the flow at a fixed position. The line formed illustrates the flow (Figure 2.6).

- *Path Lines*. Small reflecting tracer particles are added to the flow and a photographic plate exposed for several seconds.

- *Time Lines*. An electric potential is applied in pulses across a pair of electrodes installed in the liquid. In water, electrolysis will separate hydrogen from oxygen and release small H_2 bubbles at the cathode traveling with the flow. The deformation of the line illustrates the flow. Alternatively, phosphorescent dye can be added to the fluid and a laser beam used to write arbitrary patterns into the flow (*flow tagging*).

[2]This definition conflicts with the definition given by some manufacturers of measurement equipment who would like to see their techniques as non-intrusive.

When using tracer particles, not only qualitative information can be obtained about the flow but also quantitative information. Two methods for reconstructing the velocity field are well-established.

In *Laser Doppler Anemometry* (LDA) a laser beam is successively focused on different locations in the flow which are defined by the grid to be obtained by the measurement. The laser light reflected from the (moving) tracer particles contains a Doppler shift which can be used to compute the particle's velocity.

In *Particle Image Velocimetry* (PIV) two images of the flow are taken in rapid succession. If the time span between the exposures is adequate, the majority of the particles of the first image will also be found on the second image, albeit at different positions. The image plane is then sub-divided into so-called *interrogation areas* including the images of several particles. It is assumed that the particles within an interrogation area exhibit uniform movement, i.e. the local flow behavior in an interrogation area is characterized by a single vector. Each interrogation area can be considered as a discretized 2D function with pixel values (assuming grayscale images) representing function values. The sought-after displacement (and, accordingly, velocity) vectors can then be determined by cross-correlating the two functions of a PIV image pair. A peak is obtained when the most probable displacement is found. By simultaneously using two cameras with different viewpoints also a third 3D velocity component can be determined (Stereo PIV).

In practice, both approaches are used since both have their pros and cons. LDA exhibits high temporal resolution (about 1 kHz) but fails to be able to simultaneously measure the velocity of several particles (and, thus, can only be used to measure highly-reproducible flows). On the contrary, PIV has a low temporal resolution (15 Hz) but it records the flow information of the entire image plane captured by the cameras [13].

If higher accuracy is required, non-intrusive techniques must be considered. In these techniques, instead of adding material to the fluid, changes in density are taken advantage of to visualize the flow. One possibility is to project parallel light through the flow. Density changes will result in changes of the fluid's refraction index which in turn results in lighter and darker areas when exposing a photographic plate. These visualizations—known as *shadowgraphs*—are usually weak in contrast. However, when an artificial sharp light/dark boundary, e.g., a knife blade, is inserted between the light source and the flow blocking half of the light emitted by the light source, the mere light redistribution of the shadowgraph technique is complemented by the interactions of the refracted light rays and the opaque obstacle. This results in an image of higher contrast depicting density gradients. The technique is known as *Schlieren photography*.

Alternatively, a beam splitter can be used to split the light used for highlighting the flow into two parts, one of them passing through the flow before exposing the photographic plate, the other one hitting the plate directly. Since density changes

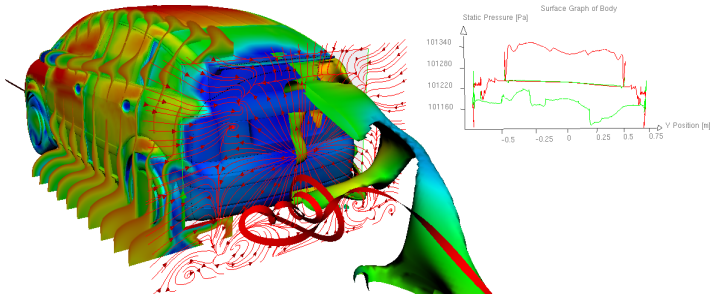

Figure 2.7 Selection of numerical flow visualization techniques compiled with the commercial flow visualization package *PowerVIZ*. Image courtesy science+computing AG.

also result in phase shift, the light when re-combined on the photographic plate will exhibit interference patterns. The technique is thus known as *interferometry*.

This work deals with numerical flow visualization so physical flow visualization techniques themselves are of minor importance. However, many numerical flow visualization techniques are inspired by physical flow visualization techniques and while physical flow visualization can start from scratch, numerical flow visualization requires a data provider which may be either computational fluid dynamics (CFD) or the very physical experiments. In Chapter 5, both LDA and PIV data are used for evaluating the implementations.

2.3.2 Numerical Flow Visualization

Particle-Based Visualization Techniques

All natural processes are time-dependent; thus, physical flow visualization exclusively deals with unsteady flows and especially with the tracking of particles (since these eventually make up the flow). On the contrary, in computer-aided flow visualization time is just another dimension that can be discarded if desired. In practice, many numerical flow visualization implementations restrict themselves to steady flows, i.e. they process a single snapshot of the flow at a certain instance $t = t_0$. This self-imposed restriction is motivated by the fact that even a single time step often comprises hundreds of millions of data values. Assuming 10^8 values per time step, an animation of just 10 s at 25 fps thus results in a total of 2.5^{10} data values. When stored with 32 bit accuracy, this amounts to 100 GB of data. Handling these data sizes poses great difficulties even on today's computing hardware. Recent work on large data visualization therefore resorts to cluster computers for the visualization mapping (Section 2.1) and uses the desktop PC simply for viewing the visualized data [16].

Discarding the time dimension simplifies certain aspects, too. When a particle is traced in a time-varying flow field starting at time t_0 at position $\mathbf{x}_0$, it is irrelevant what the velocity $\mathbf{v}(\mathbf{x}_0, t_1)$ is once the particle has reached position $\mathbf{x}_1$ at time $t = t_1$. In contrast, when physically generating streak lines by continously releasing dye at a fixed position $\mathbf{x}_0$, it *is* relevant how the velocity at position $\mathbf{x}_0$ changes since the new dye will advect along the new velocity vector. If, however, the vector field is constant in time, i.e. $\mathbf{v}(\mathbf{x}_0, t_0) = \mathbf{v}(\mathbf{x}_0, t_1)$, then the newly inserted dye will exactly follow the path of the dye released so far. Obviously, this path also matches the path followed by a particle; thus, in steady flows path lines and streak lines are identical. Furthermore, it is obvious that in any point of the particle path the path direction is parallel to the velocity vector in this point (since the velocity defines the path direction). Curves with this property are called *stream lines* in the terminology of flow visualization.

To numerically simulate stream lines an initial value problem of an ordinary differential equation (ODE) must be solved which directly follows from the definition of velocity as the rate of displacement:

$$\begin{aligned} \mathbf{x}(0) &= \mathbf{x}_0 \,, \\ \frac{d\mathbf{x}(t)}{dt} &= \mathbf{v}(\mathbf{x}(t)) \,. \end{aligned}$$

By reformulating the second equation and by replacing differentials by finite differences we obtain:

$$\begin{aligned} \Delta \mathbf{w} &= \mathbf{v}(\mathbf{w}_i) \cdot \Delta t \,, \\ \mathbf{w}_{i+1} - \mathbf{w}_i &= \mathbf{v}(\mathbf{w}_i) \cdot \Delta t \,, \\ \mathbf{w}_{i+1} &= \mathbf{w}_i + \mathbf{v}(\mathbf{w}_i) \cdot \Delta t \,, \end{aligned} \tag{2.8}$$

where $\mathbf{w}_i$ denotes the approximation of the actual particle positions $\mathbf{x}_i$ that results from the numerical integration. For $t = t_0$ the particle position is known exactly; thus, $\mathbf{w}_0 = \mathbf{x}(t_0) = \mathbf{x}_0$. It is seen that the particle path can be iteratively computed by, first, determining (interpolating) the particle velocity at its current position and, second, by integrating along the direction dictated by the velocity vector. The choice of the step size Δt in this so-called *Euler method* is a highly difficult task and the resulting particle traces will, in general, become very inaccurate when computed for a large number of time steps for flows exhibiting paths of high curvature.

Equation 2.8 is easily derived from a 1st order Taylor expansion. In *Runge-Kutta methods*, approximations of Taylor polynomials of higher order are used to obtain more accurate results. Reconsidering Euler's method geometrically it becomes clear that any inaccuracies from this method result from the inaccurate direction along which the particle position is integrated. Thus, by improving this

direction, a more educated guess for the next position can be obtained. Runge-Kutta methods therefore examine the velocities at a number of intermediate positions and compute the final approximation $\mathbf{w}_{i+1}$ using a weighted average of these velocities.

Obviously, the more intermediate samples are taken into account, the more accurate the approximation of the particle position will be. This fact is exploited by the *Runge-Kutta-Fehlberg method* to solve the step size problem by computing and comparing two guesses for the correct solution using two different-order Runge-Kutta methods. If the guesses agree, the approximation is accepted, otherwise the step size is reduced and the test performed again. Thus, the Runge-Kutta-Fehlberg method always chooses a step size that, on the one hand, is small enough for guaranteeing that a certain error threshold is not exceeded, but that, on the other hand, efficiently passes regions of low path curvature where a smaller step size would unnecessarily slow down the particle trace. The following equations have been adapted from [20]:

$$\begin{aligned}
\mathbf{w}_{i+1} &= \mathbf{w}_i + \frac{25}{216}\mathbf{k}_1 + \frac{1408}{2565}\mathbf{k}_3 + \frac{2197}{4104}\mathbf{k}_4 - \frac{1}{5}\mathbf{k}_5 \,, \\
\widetilde{\mathbf{w}}_{i+1} &= \mathbf{w}_i + \frac{16}{135}\mathbf{k}_1 + \frac{6656}{12825}\mathbf{k}_3 + \frac{28561}{56430}\mathbf{k}_4 - \frac{9}{50}\mathbf{k}_5 + \frac{2}{55}\mathbf{k}_6 \,.
\end{aligned}$$

In these equations, $\mathbf{w}_{i+1}$ and $\widetilde{\mathbf{w}}_{i+1}$ denote the approximations obtained by a 4th and a 5th order Runge-Kutta (RK45) scheme, respectively. The coefficients $\mathbf{k}_1$ to $\mathbf{k}_6$ are given by the following expressions:

$$\begin{aligned}
\mathbf{k}_1 &= h\,\mathbf{v}(\mathbf{w}_i) \,, \\
\mathbf{k}_2 &= h\,\mathbf{v}\left(\mathbf{w}_i + \frac{1}{4}\mathbf{k}_1\right) \,, \\
\mathbf{k}_3 &= h\,\mathbf{v}\left(\mathbf{w}_i + \frac{3}{32}\mathbf{k}_1 + \frac{9}{32}\mathbf{k}_2\right) \,, \\
\mathbf{k}_4 &= h\,\mathbf{v}\left(\mathbf{w}_i + \frac{1932}{2197}\mathbf{k}_1 - \frac{7200}{2197}\mathbf{k}_2 + \frac{7296}{2197}\mathbf{k}_3\right) \,, \\
\mathbf{k}_5 &= h\,\mathbf{v}\left(\mathbf{w}_i + \frac{439}{216}\mathbf{k}_1 - 8\mathbf{k}_2 + \frac{3680}{513}\mathbf{k}_3 - \frac{845}{4104}\mathbf{k}_4\right) \,, \\
\mathbf{k}_6 &= h\,\mathbf{v}\left(\mathbf{w}_i - \frac{8}{27}\mathbf{k}_1 - 2\mathbf{k}_2 - \frac{3544}{2565}\mathbf{k}_3 + \frac{1859}{4104}\mathbf{k}_4 - \frac{11}{55}\mathbf{k}_5\right) \,.
\end{aligned}$$

To compute the particle trace, the step size h is initialized to some user-specified value. With this step size, both guesses $\mathbf{w}_{i+1}$ and $\widetilde{\mathbf{w}}_{i+1}$ are computed to estimate the error. If the error is above a given threshold ϵ, h is replaced by $h' = qh$,

where q is defined as:

$$q = \sqrt[4]{\frac{\epsilon h}{2\,|\widetilde{w}_{i+1} - w_{i+1}|}}\,.$$

RK45 integration using the above equations is utilized in Section 5.5.3[3].

Computing particle traces is an iterative process, i.e. when a new position has been computed the velocity at this new position has to be determined in order to make the next step. This again involves data interpolation using, however, an updated set of data points. Tracing the particle therefore requires subsequent determination of enclosing cells. In uniform grids this operation reduces to a truncated division of the particle coordinates by the grid point distance. In unstructured grids, however, this *point location* becomes complex and requires resorting to 3D spatial data structures [44]. Since this pattern is found for most visualization algorithms, to get best visualization performance the used grid type should be as simple as possible. We will revisit this issue in Chapter 4.

Other Classical Visualization Techniques

As seen in Figure 2.7 near the rear of the car body, particle traces illustrate only a very minor part of flow and a global understanding of the flow cannot be gained unless particle traces are combined with other visualization techniques. A plethora of methods has been developed to fill in this gap. However, the focus of this work is on the acceleration of numerical flow visualization rather than on the development of novel flow visualization techniques; thus, we will restrict the discussion to the techniques used and implemented in the forthcoming chapters (especially Section 5.5.3) and refer to [69, 70, 107] for extensive surveys of flow visualization methods.

One way to get a global visualization of the vector field is to simultaneously show several particle traces. Reportedly, this approach proves unsuccessful in practice since the overall 3D structure is no longer thoroughly understood when paths interweave and the display becomes visually cluttered [29]. To prevent clutter, particle paths can be projected onto a plane as shown in Figure 2.7 but this still shows the flow only at selected positions. Recent flow visualization tools thus often resort to an alternative technique known as *Line Integral Convolution* (LIC) [8].

The LIC algorithm proceeds by initializing an input pixel field with white noise. For each pixel of the field a particle trace of a user-specified length is then

[3]Studying integration errors in relation to interpolation errors reveals that an efficient RK23 scheme already provides suffcient accuracy when using trilinear interpolation to compute off-grid particle positions [101]. The choice of RK45 integration is thus not optimal from a performance point of view.

computed by integrating along the velocity field defined by a cutplane through the initial 3D vector field. When integrating in both the direction dictated by the velocity field and its reverse, a path is obtained with the pixel defining its center. This path is interpreted as a discretized function which is then convoluted with a Gaussian function to yield the final pixel value. The contribution of a pixel to the final pixel value is thus the lower, the greater its distance to the currently processed pixel when measured along the particle path. Alternatively, a constant filter kernel, a box filter, can be employed for the convolution which allows for reusing calculation results among pixels on the same particle path and, accordingly, to speed up the image generation [90]. In contrast to classical particle traces, LIC images illustrate the local flow behavior in every pixel of the cutting plane.

Scalar fields accompanying vector fields—pressure, temperature, density—are most commonly visualized with either cutting planes, i.e. slices illustrating the magnitude of the respective scalar by the color of the corresponding pixel, and isosurfaces. We will elaborate on the latter in the context of volume visualization (Section 2.4.2).

Feature-Based Visualization

All flow visualization methods shown so far are exploratory, i.e. the flow analysis is done interactively by a researcher. Depending on the skill and experience of the researcher and the time invested into the analysis the researcher may succeed or might fail in revealing the flow behavior. The reason for this is that the field is sampled—by particle probes, isosurfaces, or cutplanes. And this sampling has to be coarse to prevent visual clutter so interesting areas may be missed if the sampling distance is too wide.

A far better approach is to employ algorithms automatically extracting interesting areas. Of course, the velocity field itself does not have an attribute that directly maps to the degree of interest so we are forced to visualize derived data. Extracting these data is what feature extraction methods have been devised for.

What "interesting" actually means depends on the application: when studying mixing processes interesting may mean finding areas where the mixing process is especially effective and where it has to be improved; in supersonic flight, shock waves generated by the airplane are of special interest. In any case, the velocity gradient tensor plays the dominant role in the extraction of flow features. Let $\mathbf{x} = (x_1, x_2, x_3)^T$ and $\boldsymbol{v}(x) = (u_1(\mathbf{x}), u_2(\mathbf{x}), u_3(\mathbf{x}))^T$ then the velocity gradient tensor (or *Jacobian*) of $\boldsymbol{v}$ is defined as

$$\underline{V}_{ij} = \left(\frac{\partial u_i}{\partial x_j} \right) \tag{2.9}$$

or, equivalently:

$$\underline{V} = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_3}{\partial x_1} & \frac{\partial u_3}{\partial x_2} & \frac{\partial u_3}{\partial x_3} \end{pmatrix} . \tag{2.10}$$

From an implementation point of view, the tensor is derived by computing the individual partial derivatives using, e.g., finite differences as shown in Section 2.2.1. By decomposing $\underline{V}$ into a symmetric part and an anti-symmetric part, $\underline{V}$ can be split into the strain-rate tensor $\underline{S} = (\underline{V} + \underline{V}^T)/2$ and the rotation tensor $\underline{\Omega} = (\underline{V} - \underline{V}^T)/2$. From this it is seen that $\underline{V}$ describes the local deformation of a fluid element.

From the velocity gradient tensor a number of features can be extracted and—using the eigenvalues of $\underline{V}$—the flow can be classified (swirling, attracting, decelerating, etc.) [29]. The only type of flow field feature addressed in this work are *vortices*. Although the concept of a vortex is intuitively understood from what is observed in sinks and drains in everyday life, there exists no clear definition of vortices. The following definition is often used but it is *not* constructive since it is recursive and thus only allows for the verification of vortex cores [76]:

> *"A vortex exists when instantaneous stream lines mapped onto a plane normal to the vortex core exhibit a roughly circular or spiral pattern, when viewed from a frame of reference moving with the center of the vortex core."*

The most simple method for detecting vortices uses only selected elements of $\underline{V}$:

$$\boldsymbol{\omega} = \begin{pmatrix} \frac{\partial u_3}{\partial x_2} - \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_1}{\partial x_3} - \frac{\partial u_3}{\partial x_1} \\ \frac{\partial u_2}{\partial x_1} - \frac{\partial u_1}{\partial x_2} \end{pmatrix} . \tag{2.11}$$

This quantity is called *vorticity* and obviously it is identical to the curl of the velocity field $\boldsymbol{\omega} = \nabla \times \boldsymbol{v}$, where, as usual, $\nabla = (\partial/\partial x_1, \partial/\partial x_2, \partial/\partial x_3)^T$. The curl, in turn, is a vector quantity indicating the axis and rate of rotation around a point, with its magnitude indicating the swirling strength and the vector pointing in the direction of the axis of rotation (the vortex core according to the given definition).

A more reliable guess for the core direction is obtained by determining the eigenvalues γ_1, γ_2, and γ_3 of $\underline{V}$. If there is one real eigenvalue γ_r and a complex conjugate pair of eigenvalues $\gamma_{c_{1/2}}$, then the eigenvector corresponding to γ_r points along the core direction [100]. Both the vorticity vector and the eigenvector method are used in Sec 5.5 to determine vortex core directions.

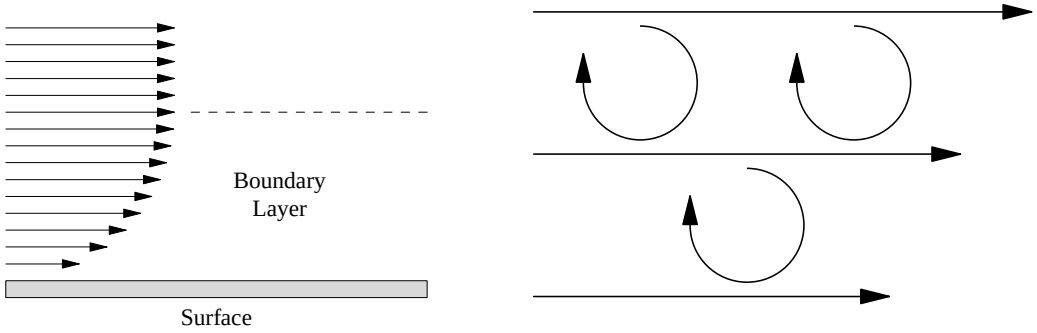

Figure 2.8 Illustration of the boundary layer (left) and the swirling motion induced by non-uniform deceleration at the object surface (right). Velocities are indicated by arrows.

The methods, however, have a common problem: while giving good estimates about the vortex core direction, they will also result in many false positives since they are only capable of detecting regions of swirling flow—which is a necessary but not sufficient criterion for vortices.

Figure 2.8, left, illustrates this problem. Fluid flowing along a solid surface is decelerated the stronger, the less the distance to the surface. The velocity diminishes when reaching the surface. This non-uniform deceleration causes shear stress which in turn induces swirling motion (Figure 2.8, right). While this swirl finally may lead to the formation of vortices when separating from the wall, the swirl itself must not be classified as a vortex by default since instantaneous stream lines in these areas do not exhibit the circular pattern demanded by the above definition.

The problem is sometimes solved by skipping boundary grid cells during the classification but this is unsatisfactory at best since the boundary layer may extend well into the interior of the vector field. Thus, better methods are needed.

An overview of the state-of-the-art in vortex detection and taxonomies for classifying the individual vortex detection algorithms is given in [35]. In this work only the predictor-corrector approach proposed in [5] and the λ_2 approach proposed in [34] will be relevant.

Predictor-Corrector Vortex Detection

The predictor-corrector algorithm proposed in [5] is a global vortex detection method basing on the assumptions that, first, local pressure minima are part of any vortex core, and, second, that vorticity gives the vortex core direction. The algorithm proceeds by first selecting a seed point by finding a pressure minimum which—according to the first assumption—is part of the vortex. At this seed point,

vorticity is then determined and a prediction for another core vertex is obtained by integrating this vector. As was argued before, vorticity does not accurately indicate the direction of the vortex core. However, even if the correct core direction were known, integrating the vector inevitably results in a predicted position off the actual vortex core due to a finite step size. To account for this inaccuracy, a corrector step is introduced which minimizes pressure in the plane containing the predicted position and perpendicular to the corresponding vorticity vector. By definition, this minimum again is assumed to be part of the vortex core. If the angle between the vector from the previous position to the position of this pressure minimum and the vector from the previous position to the prediction does not exceed a user-specified threshold, the correction is accepted and the vortex core line is traced by iterating the aforementioned steps.

In order to obtain tubular structures, a cross-section is computed for each vertex comprising the vortex core by radially sampling the pressure field in the plane perpendicular to the core line until an isovalue (which again is specified by the user) is exceeded. Naturally, this sampling results in a table mapping angles between $[-\pi, +\pi]$ to radii. This radii table can thus be considered as a 2π-periodic function which—like any periodic function—can be expanded into a *Fourier series*

$$f(x) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(nx) + b_n \sin(nx)) \ ,$$

where a_n and b_n are the *Fourier coefficients* given by:

$$\begin{aligned} a_0 &= \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)dx \ , \\ a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx)dx \ , \\ b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx)dx \ . \end{aligned}$$

The number of radial samples determines the accuracy of the Fourier coefficients, their number determines the amount of storage required for encoding a cross section. For reconstructing the cross section, the function $f(x)$ is evaluated for arbitrary angles, the number of which controls the smoothness of the final reconstructed vortex tube. That is, the given Fourier representation allows for storing the radii table with a limited number of coefficients without sacrificing the ability to reconstruct a smooth vortex surface—storage and reconstruction are decoupled.

Core line growth terminates when the cross section area approximates zero or a user-specified maximum vortex length is exceeded (preventing infinite loops

for each *remaining seed point* $\mathbf{p}_0$ **do**
  if $\mathbf{p}_0$ *not in any previous vortex* **do**
    $i \leftarrow 0$
    while *vortex skeleton continues* **do**
      (1) $\boldsymbol{\omega}_i \leftarrow$ *vorticity at position* $\mathbf{p}_i$
      (2) $\overline{\mathbf{p}}_{i+1} \leftarrow$ *predicted position by integrating* $\boldsymbol{\omega}_i$
      (3) $\overline{\boldsymbol{\omega}}_{i+1} \leftarrow$ *vorticity at predicted position* $\overline{\mathbf{p}}_{i+1}$
      (4) $\mathbf{p}_{i+1} \leftarrow$ *point of minimum pressure in the plane* $P \perp \overline{\boldsymbol{\omega}}_{i+1}$ *(correction step)*
      (5) $i \leftarrow i + 1$
    end while
  end if
end for

Figure 2.9 Predictor-corrector vortex detection algorithm as proposed in [5]. The shown pseudo-code illustrates the core line detection only and misses the vortex hull construction elaborated on in the text.

when the core runs into a spiral). Any potential seed point enclosed by the vortex tube defined this way is removed from the set of potential seed points considered for extracting the next tube, thereby successively reducing the number of potential seed points. The algorithm is outlined in Figure 2.9.

Reconsidering predictor-corrector vortex detection it is seen that the algorithm connects points where the vorticity vector $\boldsymbol{\omega}$ as defined in Equation 2.11 is parallel to the pressure gradient ∇p. In operator notation: $\boldsymbol{\omega} \parallel \nabla p$. Now let us consider the evolution of streamlines as we approach the vortex core: in a distance, they exhibit wide spiraling patterns but the closer the proximity to the core, the narrower the spirals get. Finally, when reaching the core, the spiraling vanishes yielding a streamline traveling parallel to the vortex core which means $\boldsymbol{\omega} \parallel \boldsymbol{v}$. As was shown in [66], many algorithms for computing global feature lines can be precisely mathematically defined like this using the "parallel vectors" operator.

The λ_2 Method

The λ_2 method was developed in response to the realization that intuitive measures for vortical flows like vorticity, local pressure-minima (as used in the predictor-corrector approach), and stream lines or path lines (Section 2.3.2) exhibiting circular or spiraling patterns are inadequate indicators for identifying vortices [34]. The physical effects found responsible for these inaccuracies are, on the one hand, unsteady straining resulting in pressure minima in non-vortical regions and, on the

other hand, viscous effects eliminating pressure minima in regions exhibiting vortical motion. While the former leads to false positives, the latter yields false negatives. In the derivation of the λ_2 method, therefore, terms (of the gradient taken from the Navier-Stokes equations) describing unsteady irrotational straining and viscous effects are simply discarded. In the end, the vortex detection problem reduces to an eigenvalue problem of the matrix $S^2+\Omega^2$ where S and Ω again denote the strain-rate tensor and the rotation tensor, respectively. Assuming the eigenvalues are denoted by $\lambda_1 \geq \lambda_2 \geq \lambda_3$, the λ_2 criterion defines a vortex as a connected region where two of the eigenvalues are negative or, equivalently, where $\lambda_2 < 0$ (hence the name of the method). Of course, eigenvalues are represented by real numbers. The classification of a grid point—and, thus, a vortex—resulting from the λ_2 criterion is, therefore, fuzzy rather than binary. However, since a vortex has no defined extent, i.e. since there is no intuitive diameter of a vortex tube, imposing the burden of choosing the tube diameter by selecting a suitable isovalue on the user is a sensible decision.

2.4 Volume Visualization

As seen in Section 2.3.2 current computer-aided flow visualization relies on the simultaneous display of both vector field data and scalar field data. In general, a scalar field—or *volume*—is defined as:

$$s(\mathbf{x}) = \begin{pmatrix} u_1(\mathbf{x}) \\ u_2(\mathbf{x}) \\ u_3(\mathbf{x}) \end{pmatrix} .$$

Grid cells in volume data sets are often referred to as *voxels*. Depending on whether a surface representation or a real semi-transparent volumetric object is visualized, we speak of either *indirect volume visualization* or *direct volume rendering*. Both techniques are discussed in the following.

2.4.1 Direct Volume Rendering

Direct volume rendering aims at visualizing volume data by modeling three physical effects affecting the appearance of non-opaque material: emission, caused by chemical reactions or excitation on an atomic level (as it can be observed in the glow of neon lamps), absorption, and scattering. The latter two effects are noticeable in clouds which, on the one hand, exhibit bright areas due to the scattering of sunlight but which, on the other hand, also exhibit dark areas caused by attenuation as light travels through the mixture of water droplets and ice crystals.

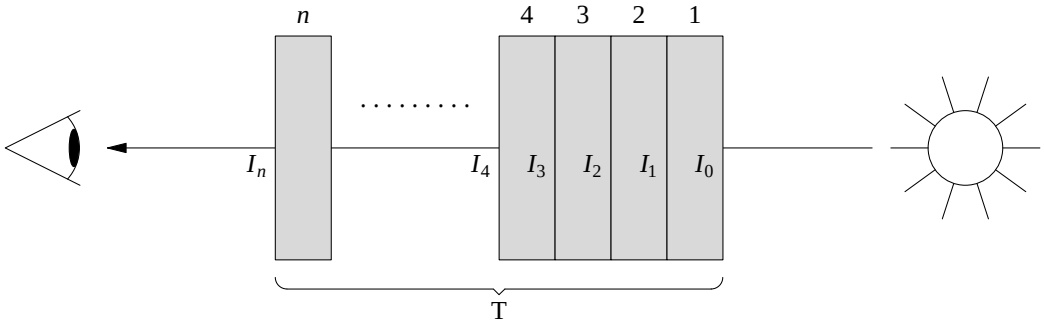

Figure 2.10 Derivation of the volume rendering integral by means of a stack of slabs of uniform optical properties.

We will restrict our discussion to absorption effects and, further, will give a simplified derivation of the so-called *volume rendering integral* governing this effect. A more mathematically rigorous exposition of various kinds of lighting models can be found in [51].

The discussion is based on the Lambert-Beer law which is also the basis for determining concentrations with spectroscopic methods. The law states that a light source of intensity I_0 when viewed through an object of thickness Δt consisting of a substance of concentration c with a wavelength-dependent extinction coefficient ϵ_λ will be perceived with the intensity

$$I = I_0 \cdot e^{-\epsilon_\lambda c \Delta t} .$$

We now think of a volume made up of n slabs S_i of equal thickness Δt made up of different materials (Figure 2.10). The respective extinction coefficients are given by ϵ_{λ_i} and the concentrations by c_i. The light emitted from the light source enters the first slab S_1 with the intensity I_0 (assuming no energy loss outside the volume). Thus, the light enters the second slab S_2 with an intensity

$$I_1 = I_0 \cdot e^{-\epsilon_{\lambda_1} c_1 \Delta t} .$$

Accordingly, the light entering the third slab has an intensity:

$$\begin{aligned} I_2 &= I_1 \cdot e^{-\epsilon_{\lambda_2} c_2 \Delta t} \\ &= \left(I_0 \cdot e^{-\epsilon_{\lambda_1} c_1 \Delta t} \right) \cdot e^{-\epsilon_{\lambda_2} c_2 \Delta t} \\ &= I_0 \cdot e^{-\left(\epsilon_{\lambda_1} c_1 + \epsilon_{\lambda_2} c_2 \right) \Delta t} . \end{aligned}$$

In general, when leaving the volume of n slabs and entering the eye:

$$I = I_n = I_0 \cdot e^{-\sum_{i=1}^{n} \epsilon_{\lambda_i} c_i \Delta t} .$$

If we now let $\Delta t \to 0$ and replace the discrete extinction coefficients and concentrations by continuous functions depending on the position within the volume and integrate over the volume of thickness $T = n \cdot \Delta t$, we obtain the intensity of the light when viewing the light source through a substance of continuously varying optical properties:

$$I = I_0 \cdot e^{-\int_0^T \epsilon_\lambda(t) c(t) dt} .$$

Since we are given a 3D volume it is disadvantageous to look-up the extinction coefficients and concentrations with the ray parameter t. Thus, we further replace this parameter by a 3D position $\mathbf{x}(t)$ and, since the light attenuation finally has to depend on the scalar value s at this position provided by the volume data set rather than (non-existent) extinction coefficients and concentrations, we combine these to a total extinction τ:

$$I = I_0 \cdot e^{-\int_0^T \tau(s(\mathbf{x}(t)))dt} . \tag{2.12}$$

The resulting Equation 2.12 can be used to generate X-ray like visualizations. If, however, we want to simulate a luminous gas with different materials, i.e. scalar values, having different colors, we need to take into account the attenuation of light leaving the respective position by using the integral derived above as a weighting factor. This introduces another integral and adjusts the integration bounds of the integral of Equation 2.12 to yield the final color $\mathbf{C}$:

$$\mathbf{C} = \int_0^T \mathbf{c}(s(\mathbf{x}(t'))) \cdot I_0 \cdot e^{-\int_0^{t'} \tau(s(\mathbf{x}(t)))dt} dt' . \tag{2.13}$$

Obviously, there is no canonical mapping from scalar values to colors and opacities; thus, both functions $\mathbf{c}(s)$ and $\tau(s)$ have to be defined by the user. The user-specified function(s) accomplishing this task is called the *transfer function* and usually can be modified at run-time to allow for an interactive exploration of the volume data set.

Exactly evaluating the volume rendering integral of Equation 2.13 is prohibitively expensive. However, it can be shown [18] that by making several approximations and simplifications, the volume rendering integral can be effectively approximated by blending semi-transparent slices derived from the initial volume in a back-to-front manner with the output color in each blending operation being defined by $\mathbf{C}^{\text{out}} = \alpha\mathbf{C} + (1 - \alpha)\mathbf{C}^{\text{in}}$. In fact, the approximation converges to the correct solution as the number of slices goes to infinity. This approach is known as *slice-based volume rendering* or *texture-based volume rendering* and it is usually implemented in graphics hardware (see Section 2.5 for the terminology). An adaptation of texture-based volume rendering is used in Section 5.3.

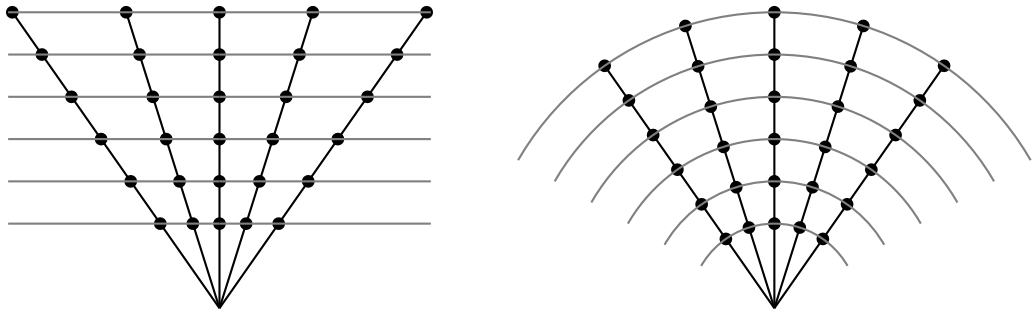

Figure 2.11 Comparison of sampling distances in slice-based volume rendering (left) and volume rendering based on ray-casting (right). Only ray-casting guarantees constant sampling rates.

Software solutions for evaluating the volume rendering integral usually resort to a different approach termed *ray-casting*. In this approach, rays are cast from a hypothetical eye or camera position into the volume. Usually, exactly one ray per pixel is generated by mapping 2D pixel coordinates to 3D world coordinates and setting up parametric ray equations based on the viewer position and the derived world coordinates. By successively incrementing the ray parameter, the volume can then be sampled at regular intervals. Compared to slice-based volume rendering, the latter is a major advantage since it reduces artifacts (Figure 2.11). The compositing, however, now has to be accomplished in a front-to-back order which makes the blending operation more expensive: $\mathbf{C}^{\text{out}} = \mathbf{C}^{\text{in}} + (1 - \alpha^{\text{in}})\alpha\mathbf{C}$; $\alpha^{\text{out}} = \alpha^{\text{in}} + (1 - \alpha^{\text{in}})\alpha$. We will have more to say about the pros and cons of the two approaches in Section 5.4.2.

In both ray-casting and slice-based volume rendering the volume is sampled *before* the transfer function is applied. Thus, while the sampling frequency might suffice for the original volume data, it might no longer suffice after the transfer function has been applied (as, e.g., spikes might have been introduced). *Pre-integration* is an approach developed to address this problem [18]. In this technique, the volume rendering integral is calculated in advance for pairs of sample values. This calculation is performed in software and only for a relatively low, fixed number of combinations. Thus, high-quality integration techniques with adaptive sampling rates can be utilized to compute the contributions of the slabs defined by the two samples. The price paid for the improved quality is the need to sample the volume both on the front and back plane of the slab during rendering and the need to re-compute the pre-integration table after changing the transfer function. Pre-integration is used in Section 5.4.2.

2.4.2 Indirect Volume Visualization

For some applications displaying the volume as a stack of semi-transparent slices might be unwanted. For example, in medicine, when a CT scan is taken of a skull exhibiting fractures, it is most useful to show the object of interest—the bone—in its natural form, i.e. as a surface without surrounding soft tissue. Since the material comprising the skull will be represented by similar scalar values in the data set, the skull can be reconstructed by computing an *isosurface*, i.e. a surface containing only those vertices whose corresponding data values match a specified *isovalue*. This is usually referred to as *indirect volume visualization*. A well-established method for computing isosurfaces is the *Marching Cubes* (MC) algorithm [47], a fast isosurface generating algorithm designed for uniform grids.

The algorithm proceeds by processing the volume grid cell by grid cell (or, equivalently, by determining the isosurface intersection with one cuboid before marching to the next). In this first phase every grid point of the cube is assigned a value of 1 if the corresponding scalar value exceeds or equals the isovalue and a 0 if the value is less than the isovalue specified by the user. Since there are eight grid points associated with each grid cell, this phase results in an 8-bit code encoding all $2^8 = 256$ possibilities of how the isosurface can intersect the cube. For each of the 256 cases, there is a uniquely defined set of edges that are intersected by the isosurface for the respective case; thus, the edge intersections can be efficiently determined by interpreting the 8-bit code as an index which is then used for accessing the corresponding edge set in pre-computed 256-element look-up table.

Once the edge set has been found, the edge intersections are computed by linearly interpolating between the two end points of each edge. This already determines the vertices that will become part of the isosurface. In order to display the isosurface, a triangle mesh must be fitted to these vertices. In the MC algorithm this triangulation is efficiently solved by exploiting topological symmetries. If, e.g., only a single grid point lies on one side of the isosurface while the remaining seven points lie on the opposite side, it is irrelevant whether the single grid point lies inside or outside the surface (i.e. whether the corresponding bit is set or cleared). It is also irrelevant which of the eight possible grid points is the outlier. In either case, a single triangle has to be generated.

In the end, only 14 cases have to be differentiated which, by applying permutations, are then used to construct a 256-element triangle table. Each element of this table stores a set of indices to be used for accessing the vector of surface-edge intersections and, finally, for constructing the geometry that the currently processed cube contributes to the isosurface.

If not only intersection points but also grid point normals are interpolated (for the latter we can use normalized gradients computed with finite differences, Sec-

tion 2.2.1) smoothly shaded surface visualizations can be obtained.

An MC implementation is used in Section 5.5, a closely related technique, *Marching Tetrahedra*, will be used in Section 5.4.1.

2.5 The Rendering Pipeline

The above sections described the visualization mappings most relevant for this work. The primitives resulting from this stage of the visualization pipeline—line segments comprising a particle trace, triangles comprising a pressure isosurface—need to be rendered next. Like the visualization process this rendering step is also described as an abstract pipeline [102] with primitives provided by the application flowing in and an image—a set of pixels—flowing out. This section outlines the rendering pipeline, thereby introducing terms and concepts used in the following chapters.

For convenience, geometry to be rendered is usually specified in a local coordinate system. Often, this local coordinate system is chosen such that the object center coincides with the origin. To specify the spatial relationship between the individual objects, these local coordinate systems have to be combined into a single global *world coordinate system*. This world space is also where lighting computations are performed and texture coordinates are specified. In graphics terminology, the term *texture* is used to denote a one-, two-, or three-dimensional image for modifying the color of a pixel produced during rasterization (see below). *Texture coordinates* define the mapping of these images to pixels. Textures are used extensively in Chapter 5.

The world coordinate system is independent of any viewer. Thus, a third coordinate system is introduced by positioning a virtual camera in an arbitrary location in world space. This *eye coordinate system* is defined by a viewing direction, an up vector, and a right vector (assuming a right-handed coordinate system). Since the visibility of polygons is decided based on the angle between the line-of-sight vector from the polygon center to the eye position and the polygon normal, eye space is where culling is performed.

Eye space coordinates are still three-dimensional. For viewing on a screen, these 3D coordinates must eventually be mapped to 2D *screen coordinates*. This marks the final transformation from eye space to screen space. Screen space is where clipping is performed, hidden surfaces are removed in scenes of several opaque objects with different depth values, and shading models are applied to determine pixel colors.

The eye-space to screen-space transformation is non-linear and thus cannot be described in matrix notation. To allow for matrix-notation, the projection is thus usually split into a linear part and a non-linear part. The former is described

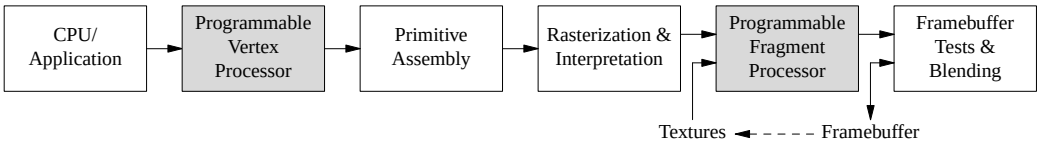

Figure 2.12 Model of a modern GPU as presented in [57]. Shaded boxes indicate freely programmable units.

by the *projection matrix*, the latter is accomplished by a perspective division by the 4th component of the homogenous coordinates (i.e. the projection matrix is the only one using the w-component in a non-trivial way by assigning it a value $\neq 1$). For perspective projection the projection matrix defines the *viewing frustum*, a closed volume anything outside of which will be clipped. Frusta definitions and manipulations of the projection matrix will play a central role in Section 3.5.

Once screen space coordinates are available, the final stage consists of rasterizing the polygons, i.e. determining which pixels are covered by the polygon. During this last step shading, texturing, and hidden surface removal are performed on a per-fragment basis. In this context, the term *fragment* corresponds to a pixel carrying additional information like texture-coordinates and depth values.

This pipeline model is convenient since it can be—and has been—directly mapped to actual graphics hardware.

2.6 Graphics Processing Units

Figure 2.12 illustrates the rendering pipeline as implemented in a modern graphics processing unit (GPU). In principle, knowledge of this pipeline is not required as long as the various stages cannot be controlled other than by changing the input data and settings various states influencing the transformation (turning lights on and off, enabling or disabling blending of semi-transparent polygons, etc.), i.e. as long as the pipeline must be considered *fixed-function*. This, however, has changed with the introduction of programmable vertex and fragment processors (marked in gray in Figure 2.12). The following sections introduce vertex and fragment programs and elaborate on how these concepts can be used for general purpose computing.

2.6.1 GPU Basics

Programmable vertex and fragment units provide the user means for partially replacing the fixed-function pipeline: the transformation and lighting usually performed by the geometry unit, on the one hand, the texturing and shading accom-

plished by the rasterization unit, on the other hand. The custom functionality is implemented as a sequence of assembly instructions. This sequence is called a *vertex program* or *vertex shader* when the target processor is the geometry unit and a *fragment program* or *pixel shader* when addressing the rasterization unit.

Like any program, vertex and fragment programs transform input data to output data. For vertex programs, the input includes the vertex position in world coordinates, the world-space normal vectors, and world-space texture coordinates. The program returns transformed vertex coordinates, texture coordinates, and color information. The latter two are often handed over unmodified. For fragment programs, the input includes color information and texture coordinates while the output comprises color and depth information (for hidden-surface removal). In addition to per-vertex and per-fragment input attributes, respectively, global parameters like transformation matrices can be passed to the programs.

Two application programming interfaces (APIs) are available for feeding data into the pipeline and for programming the vertex and fragment units. The first, OpenGL [109], is an extensible, open standard specified and designed under the auspices of a consortium comprising the major graphics hardware vendors. On the contrary, the design of Direct3D is controlled by single vendor (Microsoft). Direct3D is a component of DirectX and thus only available on MS Windows platforms. In general, both APIs provide comparable functionality. However, while OpenGL tends to provide more experimental features, Direct3D implementations are often more efficient—driven by the computer games industry for which MS Windows is the most commercially attractive platform. The graphical applications implemented for this work are all based on OpenGL. An exception is made in Section 5.3.

In addition to assembly language, both OpenGL and Direct3D provide C-like high-level languages for programming the vertex and fragment units. For OpenGL, there are both vendor-specific solutions like Cg [57] and vendor-independent solutions (OpenGL Shading Language [36]). Both languages are made available by a third party. For Direct3D, the only available high-level shader language is HLSL [65]. HLSL is an integral component of DirectX and no third-party component. In this work, Cg (Section 3.3.3), HLSL (Section 5.3), and assembly language (Sections 5.4.1 and 5.4.2) are used.

2.6.2 General-Purpose Computations

Vertex and fragment programs result in an increased flexibility which originally was intended to create advanced visual effects [22]. However, a number of GPU features recommend the hardware also for general-purpose computations.

First, data processed by a GPU are predominantly RGBA colors or homogenous coordinates, each of them having four components. Accordingly, the instruc-

tion set of modern GPUs has been designed for vector processing, performing all numerical operations in parallel for four-component vectors.

Second, to allow for high-accuracy vertex processing, some modern GPUs are able to both store and process data values with 32 bit numerical accuracy. The accuracy of GPU computations is thus comparable to CPU computations using single-float accuracy.

Third, in order to allow for real-time effects at interactive framerates, most modern GPUs include several geometry and rasterization units. For example, the ATI 9800 XT graphics board has 16 rendering pipelines; thus, current GPUs are powerful multi-processors made accessible by programmable vertex and fragment units.

Finally, while first-generation GPUs lacked dynamic looping and branching—two fundamental requirements for general-purpose computing—and imposed rigid limitations on the maximum number of instruction slots and registers, these features have become available with the advent of DirectX Pixel Shader 2.x/3.0 [53] and the OpenGL `NV_fragment_program2` [58] extension. Modern GPUs thus not only have the required capabilities for general-purpose computing; they also have the computational power for making them an attractive platform for general computations.

GPU-based implementations, however, will not be advantageous for arbitrary algorithms. The architecture of modern GPUs resembles a *Single Instruction, Multiple Data* (SIMD) architecture, i.e. the GPU can perform a vast number of computations in parallel for each vertex and fragment, respectively, as long as the algorithm for the computation is identical for all entities to be processed. In addition, since both the maximum number of input values that can be read (by, e.g., texture look-ups) as well as the maximum number of output values that can be written are severely limited, GPU implementations will usually be beneficial if and only if the respective algorithm works on a small—or even better: fixed—amount of data. Algorithms with this property are referred to as *local* algorithms. Thus, referring to Section 2.3, computing the vorticity of a vector field is a local operation while tracing particles is not since we might need to access an arbitrarily large number of values if the trace runs into a spiral.

Reconsidering the GPU model shown in Figure 2.12, it becomes clear there is a one-to-many relationship between polygons and pixels. In order to obtain a well-balanced system, hardware manufacturers therefore equip their GPUs with considerably more fragment processors than vertex processors—NVIDIA's NV40, e.g., includes 16 fragment units but only six vertex units. The definition of the various pipeline stages therefore not only shows what part of the pipeline *can* be programmed and which functionality *can* be affected; it also tells us what pipeline component *should* be employed for best performance in general-purpose computations: the rasterization unit.

In this work, therefore, the vertex unit has not been used at all for general purpose computations. General purpose computations using the fragment unit are described in Sections 3.3.3, 5.3, 5.4.1, and 5.4.2. Details regarding the choice of the respective platform (Direct3D vs. OpenGL, NVIDIA vs. ATI) are given in the respective sections.

2.7 Stereo Viewing

When talking of 3D graphics what is usually meant is the visualization of a scene specified in 3D world coordinates displayed on a 2D screen by some form of projection (usually orthogonal projection or perspective projection). If the visualization is well-produced, it will provide various depth cues [46]. Objects are, e.g., usually displayed smaller when farther away while textures become denser. Also, objects may obscure each other, indicating that the obscuring object is closer to the viewer. But since there is only a single view of the scene the same image is presented to both eyes. Real stereo vision as it is available for actual 3D objects is, therefore, absent. For many applications the depth cues mentioned above are sufficient. Sufficient however is not synonymous to optimal. For positioning particle probes or for studying complex, interacting 3D structures real stereo vision might be advantageous and ease the task. Therefore, stereo is often used to further enhance the visualization.

To provide real stereo vision two different views must be generated, one for the left eye and one for the right eye. Each must only see the view destined for it. Thus, stereo viewing has to address two problems: first, deriving different views, and second, physically separating and presenting the views.

2.7.1 Stereo Techniques

There are two classes of stereo techniques, active stereo and passive stereo.

Active stereo uses a single display device with a high refresh rate and specialized glasses synchronized with the display. The device displays the images for the left and right eye in alternating order. When the image for the left eye is shown, the left glass is made transparent while the right glass is made opaque and vice versa. Active stereo techniques suffer from the requirement for specialized hardware and the problem of synchronizing the display device with the shutter glasses. *Passive stereo*, on the other hand, does not suffer from these drawbacks. There are many solutions.

One popular approach uses polarized light. Images for the left and right eye are generated simultaneously with different polarizations (usually by installing filters in front of the display device). The users wear glasses with polarization filters.

Passive stereo with polarized light is available as a single-projector solution using a so-called *z-screen* which switches the polarization electronically by modulating a liquid crystal layer (requires high refresh rates) or a two projector solution where static polarization filters are installed in front of the lenses and each projector displays the images for one eye exclusively.

ChromaDepth [4] is a proprietary technology taking advantage of the fact that on a black background blue will appear farthest to the viewer while red will appear closest. The remaining colors will appear farther the greater the hue when representing the color in HSV color space. The drawback of this solution is the need to alter the scene colors, a requirement out of the question for scientific visualization where color often represents the magnitude of some quantity.

Another widespread approach is *anaglyph stereo*. Assuming the scene colors are represented by a set of primary colors $\mathcal{P}$ the images for the left and right eye are rendered with subsets $\mathcal{L}$ and $\mathcal{R}$ of $\mathcal{P}$ with $\mathcal{L} \cap \mathcal{R} = \emptyset$. Accordingly, the sets $\mathcal{L}$ and $\mathcal{R}$ can be rendered into the same image without interfering with each other and separated again using colored glasses of the respective colors. To represent the complete color space we further demand that $\mathcal{L} \cup \mathcal{R} = \mathcal{P}$. In the common RGB color space, red/green therefore is an unsuitable choice since it lacks the capability to represent blue tones while red/cyan is a suitable choice. Viewing anaglyphs usually results in imperfect color perception since pure red, e.g., is only seen by one eye; thus, the color—when fused by the brain—is perceived darker than it actually is. On the other hand, the solution requires only a single projector and cheap glasses, properties recommending it to this work and outweighing its drawbacks. Anaglyph stereo is used in Sections 3.5 and 5.5.3.

2.7.2 Frusta Calculations

As seen in Section 2.5 the viewing frustum determines what part of the scene is seen. For stereo viewing, two frusta are defined. Each frustum corresponds to a single eye, with the center of projection defining the eye position. In this configuration, points out of focus project to different positions on the projection plane. The signed distance between the projection when viewed with the right eye and the projection when viewed with the left eye is called the point's *parallax*. For points in front of the projections plane (i.e. between the viewer and the projection plane) the parallax is negative, for points behind the projection plane the parallax is positive. The parallax is zero for points lying on the projection plane. Two methods are popular for deriving the frusta: the toe-in method and asymmetric frusta (Figure 2.13).

The *toe-in* method uses two symmetric frusta and rotates them in opposite directions in order to focus on the same point in space. While this approach leads to the desired horizontal parallax it also results in vertical parallax with perspective

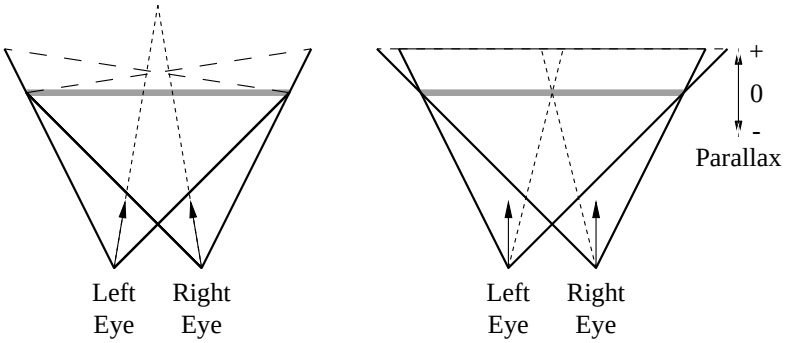

Figure 2.13 Alternatives for computing stereo views. Left: Incorrect toe-in method resulting in vertical parallax; right: correct method using asymmetric frusta. The stereo window (projection plane) is shown in gray.

projection since the projection planes are no longer perpendicular to the viewer. Vertical parallax is uncomfortable for the viewers.

A better alternative keeps projection planes perpendicular to the camera. However, if the eyes are separated and the scene must be projected onto the same region this means that the frusta have to be sheared and thereby lose their symmetries. This method is, therefore, known as *asymmetric frustum projection*. We will give a short overview of how to derive the projected point coordinates for asymmetric frusta [2].

We assume the camera (center of projection) to be located at $(0, 0, d)$ looking in the direction of the negative Z-axis. The projection plane is assumed to be the XY-plane. A point (x, y, z) then is projected to (kx, ky) where $k = d/(d - z)$.

For the left-eye and right-eye frusta we need to shift the center of projection to the left and to the right, respectively, by half the eye distance $c/2$. Equivalently, we can move the scene to right and to the left, thus obtaining $x_{\text{left}} = x + c/2$ and $x_{\text{right}} = x - c/2$. The new projections are then obtained by (kx_{left}, ky) and (kx_{right}, ky), respectively. Using these modified expressions, however, a point $(x, y, 0)$ on the projection plane no longer retains its position but is shifted horizontally (or, put another way, points on the zero-parallax plane no longer exhibit zero parallax). Considering just the projected X-coordinate x' we see that:

$$x' = k\left(x \pm \frac{c}{2}\right) = \frac{d\left(x \pm \frac{c}{2}\right)}{d - z} = x \pm \frac{c}{2} .$$

To account for this horizontal shift we need to introduce a correction $\Delta = \mp c/2$. The projected point coordinates (x_p, y_p) are then given by

$$(x_p, y_p) = \left(k\left(x \pm \frac{c}{2}\right) + \Delta, ky\right) . \tag{2.14}$$

In this work, Equation 2.14 is used in Section 3.5 and Section 5.5.3. The former section elaborates on how asymmetric frusta can be derived for a given monoscopic view specified with the OpenGL API.

2.8 Data Display

In the final stage of the visualization pipeline the images resulting from the visualization mapping and rendering are usually displayed in a window provided by the host's window system. On Unix-based systems the dominant window system is the *X Window system*, the dominant graphics API OpenGL (Section 2.6.1). The glue connecting these components is called *GLX*.

2.8.1 The X Window System

The X Window System [61] (or simply *X*) is a de-facto standard for the development of multi-window applications with graphical user interfaces. Its design is controlled by a consortium of major hardware vendors and research institutions. As a result, X is portable, i.e. graphical applications can be written for a number of different systems, requiring only minor modifications or no modifications at all.

The X Window System is client-server based. Clients are called *X clients*, servers are called *X servers*. On each host there may be several clients and several servers. X is network-oriented which means that clients and servers may originate from different vendors and may be connected by a variety of communication channels, including both network connections (TCP, DECnet) or some local interprocess communication protocol when running on the same host. From a user's point of view this is completely transparent. This network-transparency of the X Window System is exploited in Chapter 3.

In X terminology, the server is the program controlling the display (a workstation with input devices and screens). Clients, on the other hand, denote graphical applications.

The major tasks of the server are handling client accesses, interpreting client messages coming in, two-dimensional drawing (of both graphics and text), and generating outgoing messages. The latter is required when, e.g., user input has occurred. Messages informing about user interactions are called *events* and include notifications about key and mouse button presses and releases, window movements, etc.

The main task of clients is to specify to the server what the application windows and their contents should look like and to process any events relevant to the application. The X server will then take care of the actual drawing. Clients give

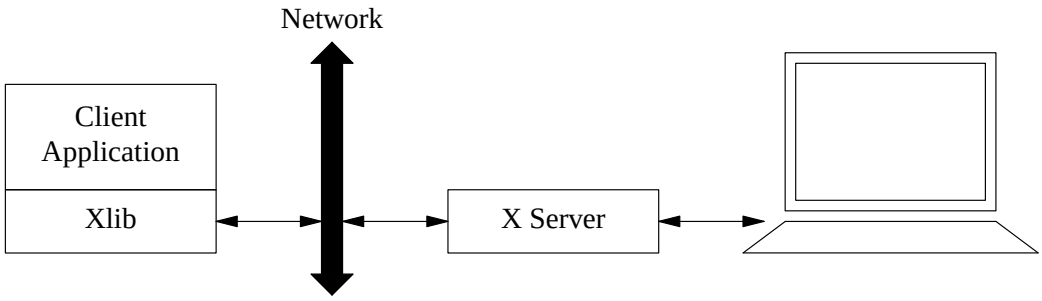

Figure 2.14 Clients and servers in the X Window System as presented in [61].

only hints about the size and position of windows, actual dimensions and positions are set by the *window manager* which is not an integral part of X. Window resizing and stacking are also handled by the window manager. In rare cases, the functionality provided by the window manager must be suppressed (Section 3.2).

Communication between clients and servers is done by exchanging X protocol requests [60] (Figure 2.14). The protocol includes requests for the entire lifecycle of a window: initiating the connection to the server (known as *opening a display*), creating windows, filling the window with text and graphics, processing events, destroying windows, closing server connections. Client-side requests are generated with the help of the *Xlib* library [61], a collection of C functions giving unconfined access to the functionality the X server provides. Xlib is the lowest level of X programming and thus often tedious to use. Therefore, several toolkits are around for making the task of creating windows and user interface elements more comfortable, e.g., Qt and the OpenGL Utility Toolkit (GLUT). In this work only GLUT is used which—as the name implies—is a limited but ubiquitously available toolkit for creating windows to be used in conjunction with OpenGL for 3D graphics.

No system as complex as the X Window System provides the functionality satisfying every user. X therefore has been designed to be extensible and it provides means for adding features to an X server without the need to modify the server code. In this work two extensions are used: the MIT shared memory extension for high-speed exchange of image data between client and server and the GLX extension described in the next section.

2.8.2 GLX Basics

The X Window System only provides very rudimentary functions for drawing graphics (arcs, polygons, rectangles, etc.). For sophisticated 3D graphics required for scientific visualization this is insufficient. Therefore, one, has to resort to

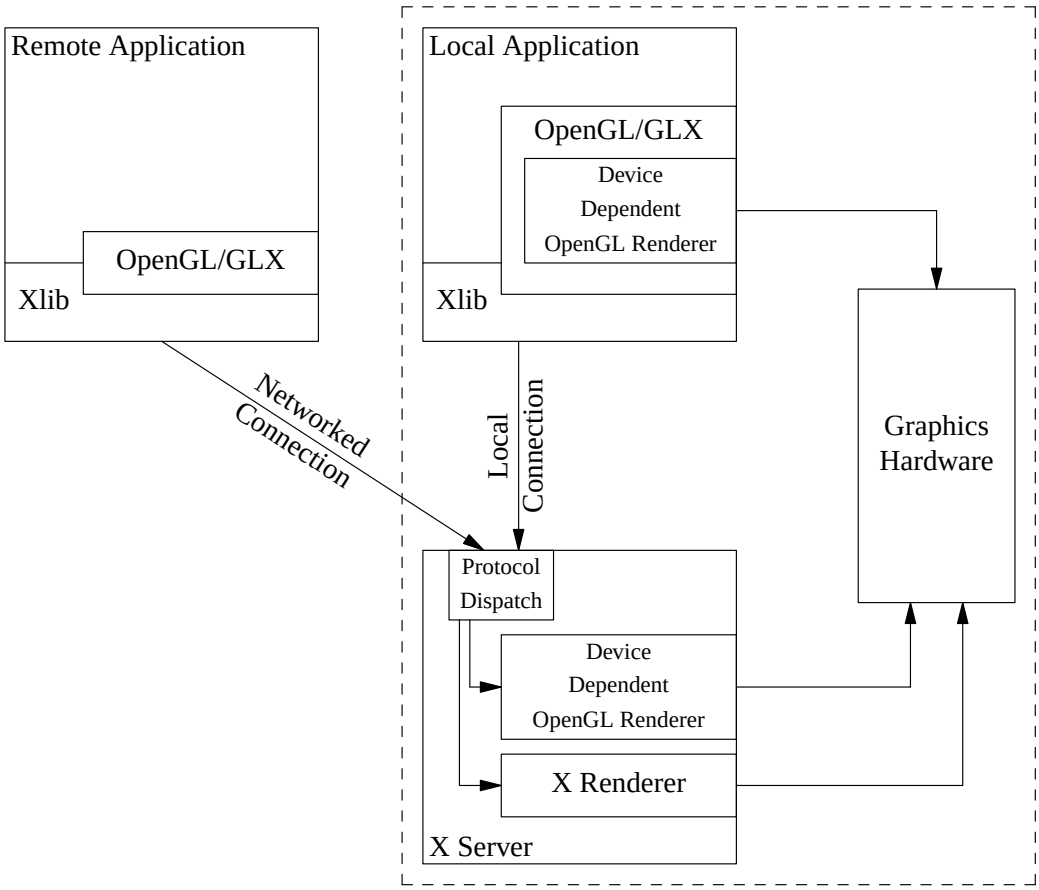

Figure 2.15 Architecture of the GLX extension as presented in [37].

more advanced APIs, preferably OpenGL (Section 2.6.1). OpenGL, however, is meant to be portable and, accordingly, the specifications lack any references to specific windowing systems. Some platform-specific "glue", therefore, is needed to integrate OpenGL into the window system. On MS Windows systems, this glue is called *WGL*. WGL is not relevant for this work. On Unix-based systems, the task is accomplished by GLX [37]. GLX will be extensively used in Chapter 3.

GLX is implemented as an extension to the X protocol. It allows for creating a GLX context which in turn can be used to issue OpenGL calls. Two operating modes are supported (Figure 2.15). *Direct mode* circumvents the GLX protocol and permits the client to directly communicate with the hardware. This is known as *direct rendering* [39]. In contrast, since GLX is an X protocol extension it is naturally network-transparent (like any other X extension) and thus allows for issuing OpenGL calls on remote hosts. Thus, when a display is opened on a host connected to the local machine by some network, OpenGL commands are en-

coded using the GLX protocol and transmitted to the remote host as part of the standard X protocol byte stream. The data is decoded by the receiving X server and the OpenGL commands directed to the remote graphics hardware, thereby ignoring the local graphics resources. This mode, independent of whether the GLX requests are transmitted over a network connection or some local inter-process–communication channel, is known as *indirect mode* or *indirect rendering*.

All OpenGL rendering is done with respect to a *rendering context*, a virtual OpenGL machine independent of any other contexts. A *GLX context* is a handle to a rendering context consisting of a client state and a server state.

As seen in Section 2.5, the rendering of complex scenes is computationally expensive. For many applications, especially those showing animated scenes or those allowing the user to interactively manipulate (rotate, translate, scale) the objects comprising the scene, displaying intermediate results should be avoided. This issue is independent of the framerate (images per time unit) and the monitor refresh rate since in any case incomplete images might be shown. GLX provides means for alleviating this problem by assembling the image in a background buffer. Once the image is completed, the background buffer and the front buffer (seen by the user) are switched. This technique is known as *double buffering* and it will become relevant in Chapter 3.

2.9 Dynamic Linking and Loading

A final technical topic of relevance for this work is not visualization-related at all but rather a fundamental operating system concept. When building an executable application from a number of source code files, the individual files are first compiled to object files and finally bound to a single executable. The former part is accomplished by the *compiler*, the latter by the *link editor*. Since one intention for distributing the source code of an application over several files is code reuse, functions in one source file will usually reference functions defined in several other source files. Connecting these symbolic references of one object file to symbolic definitions of another object file (a process known as *binding*) is the task of the *linker*. The final output may have several forms. For the problem of remote visualization, only three forms are relevant. So-called *static executables* are ready-to-run processes with all dependencies resolved by including the required functionality in the executable. *Dynamic executables*, on the other hand, depend on one or several external object files in order to resolve dependencies and to become a ready-to-run process. *Shared objects*, finally, are collections of object files that can serve as functionality provider for dynamic executables. The *dynamic linker* is the program combining a dynamic executable and shared objects into a runnable process. This is known as *dynamic linking*.

The benefits of dynamic linking are manifold, the main benefit is code reuse for 3rd party software. A programmer wishing to make available some functionality simply creates a shared object—a *library*—with the respective functions. Anyone wishing to use the functionality then just links against the library. When malfunction in the library is detected, the error can be fixed and the library reinstalled. Since the libraries are not part of an executable, any application using the library will automatically benefit from the updated functionality when started anew. Of course, when the applications agree on where to locate library functionality, disk space can be saved by storing only a single copy of the file. Similarly, when agreeing on the memory address of a library once it has been loaded, main memory usage can also be made more efficient.

Two advanced linking features are of special importance: first, to be able to specify libraries that are to be loaded before all libraries specified by the executable (termed *pre-loading* in, e.g., Linux and Solaris) and, second, the ability to recover functionality of libraries overridden by the mechanism mentioned first. The latter feature is known as *run-time linking* [31, 48]. In combination, pre-loading and run-time linking allow for the selective modification (or even replacement) of functions used by an executable.

From an implementer's point of view, pre-loading starts by identifying the libraries relevant for an application using some tool for determining shared library dependencies (on Linux systems `ldd(1)`[4]). Naturally, each library provides a list of function prototypes defining the library interface. The most difficult task is the definition of a set of functions to be overridden for attaining the desired behavior. Once the relevant functions have been identified, a custom shared object file is created defining functions declared identical to the functions requiring modifications. When running the application, the pre-load library is usually specified by setting an environment variable (`LD_PRELOAD` in Linux and Solaris). The operating system now guarantees that whenever the overridden function is called, the new functionality will be invoked instead of the original one. If only minor modifications of the functionality are required, a pointer to the original function is recovered in some initialization part or during the first call to the respective function. Using this pointer and—if required—the arguments passed to the overridden function, the original function can then be invoked. The most common programming interface for run-time linking is defined by the functions `dlopen(3)` and `dlsym(3)`.

Pre-loading and run-time linking will be used throughout Chapter 3 for a number of different applications.

[4] As usual, numbers in brackets denote the section of the Unix manual pages documenting the respective command or library function.

Chapter 3

Remote Visualization

There are two sources of flow data in numerical flow visualization: physical flow measurements and computer-aided numerical simulations. The amount of data generated in physical experiments is usually small since, one the one hand, single-point measurement techniques like LDA require successive accurate, time-consuming re-positioning of measurement devices. On the other hand, multi-point measurement techniques like PIV still at best generate 2D slices of the flow field with 3D velocity information. In any case, obtaining time-dependent data sets is only possible in special configurations where the flow is periodical and where nearly identical flow field states can be reproduced repeatedly [43]. But even then the complexity of the measurements prevents the generation of data sets in the gigabyte range. Analyzing these data, therefore, usually poses no problems using the hardware generally available to the researcher at his workspace.

In contrast, computer simulations tend to produce more and more data often investing months of supercomputing power. The results are often in the terabyte range (see, e.g., [16]). Analyzing these data locally can be difficult for several reasons. If the local hardware resources (main memory, CPU and GPU power, supported features) are insufficient, local analysis is impossible at all. This is the most common hindrance. If the local hardware suffices but the simulation parameters change frequently and the network bandwidth is low, the data transfer may become very time-consuming and tedious; thus, the researcher might be reluctant to visualize data remotely. If the network allows for fast data transmission and local resources are sufficient for processing the data, local analysis is possible. Replicating data sets on the hosts of researchers analyzing these data, however, is inefficient since it multiplies the overall memory requirements and unnecessarily raises workstation costs, giving remote visualization an economical aspect. Finally, in industrial applications where new designs are evaluated, the raw simulation data may be highly confidential so a company might restrict the storage location of the data to computers in the local network. On the other hand, im-

ages mainly carry qualitative information and are, therefore, not as sensitive as numerical data. While the latter two issues are pleasant features of using distant resources for generating the visualization, the former two are obviously capable of accelerating the flow visualization and of making the data analysis task more efficient.

As long as the server is Unix-based, a simple and cost-effective remote visualization solution is to use the remote rendering capabilities of OpenGL in conjunction with the GLX extension to the X server as seen in Section 2.8.2. The benefit of this approach is its application-independence and generic nature, allowing for the use of any graphics application without requiring any modifications of the source code. The drawback of the solution, however, is the fact that only few GLX-capable X servers are available for non-Unix platforms (the Hummingbird X server with the Exceed 3D add-on being a rare exception). Even worse, remote visualization based on GLX does not take advantage of high-performance graphics resources of the render server but essentially only of CPU power and main memory. Thus, if only the remote machine offers the required amount of texture memory or special features (both were issues especially in the early days of computer graphics), GLX is no adequate solution.

From a remote visualization point of view forwarding OpenGL commands is, therefore, exactly what is *not* desired. As a result, a new research field termed *remote visualization* was born whose goal is to provide a visualization service to a lean client by exploiting the computing and graphics resources of a powerful remote server.

3.1 Existing Remote Visualization Solutions

In [49], a solution for visualizing time-varying volume data over wide-area networks is presented. The system involves a dedicated display interface and a daemon for compressing, transferring, and displaying image data and for providing means for communicating user input back to the renderer. The data is transmitted via a custom transport method which allows for the incorporation of arbitrary compression algorithms. No significant use of local graphics hardware is made.

In contrast, the *Visapult* system [7] promotes the idea of *shared rendering* where some visualization is accomplished using the remote hardware and the final image is obtained with the help of local graphics resources. The concept is illustrated by the example of a remote volume visualization tool where the client side implements an image-based volume renderer capable of synthesizing new images based on already available views. A custom transport protocol is used for the data exchange between client and server. The system is not application transparent and—by its very definition—demands considerable graphics resources on

both the client side and the server side.

The ideas presented in [17, 19] are similar to the shared rendering approach taken in [7]. Here, a Java3D client allows for selected interaction tasks (like modifications of the transfer function) using only the local graphics hardware, thereby circumventing high-latency network transmissions. The concept is illustrated for medical volume rendering and a post-processing tool for computer-simulated crash tests. Since a major part of the viewer needs to be re-written for every application, the approach—while general to some extent—is very work-intensive.

Simultaneously to what is described in the following, two solutions of a new class of solutions have been proposed striving for application transparency. The SGI *Vizserver* [86] relies on dynamic executables and the possibility to selectively override functions as outlined in Section 2.9. Similar to the previously described solutions, a dedicated client is used to receive and display images generated by the server. Initially, both the client component and the server component were limited to SGI platforms; in the meanwhile the system has been enhanced to also support Linux, Sun Solaris, and MS Windows clients. The server component, however, is still bound to a single platform. The TGS Open Inventor 3.0, on the other hand, does not restrict the server platform and utilizes the 3rd party *Virtual Network Computing* [73] software for data display. While SGI's solution is platform-limited, the TGS solution is limited to a single application (albeit not conceptually).

None of the solutions, therefore, is generic with respect to both hardware and software. However, since generic solutions are in general slower than specialized software, the question arises why one should care about application transparency. There are several reasons. First, if applications have to be modified to make them network-aware, the respective remote visualization solutions cannot be used with undisclosed source as it is the case for commercial applications. Second, even if source code is available, non-generic solutions burden the programmer to modify (maybe) 3rd party source code which is time-consuming and error prone. Third, when using remote resources of a computing center one has to get along with whatever machines are available. If the remote visualization software is limited to a single platform its applicability is severely limited. Similarly, being forced to use a single visualization suite for all visualization tasks is out of the question—the tool will neither exhibit the performance nor the functionality of specialized software.

The following sections present the first truly generic remote visualization solution. It was first published together with M. Magallón [95] and, like the solutions by SGI and TGS, it is based upon the X Window System and its associated protocol and the concept of dynamic linking elaborated on in Sections 2.8.1 and 2.9, respectively.

3.2 X-Window–Based Image Transmission

The most basic remote visualization system based on the X Window System and OpenGL/GLX runs standard OpenGL applications and forces the issued OpenGL calls to be directed to the hardware where the application is executed. We will refer to this host as *render server*. When the rendering is finished, the framebuffer is read out and the image data along with graphical user interface (GUI) elements sent over the network using the X protocol to the machine which is operated by the user. The latter machine will be called the *interaction server*.

The benefit of this basic approach is that it takes advantage of advanced graphics features on the render server side and that it does not demand for OpenGL capabilities on the interaction server. Furthermore, since the application invocation when using the remote visualization library does not differ from a local invocation (besides setting an environment variable) this solution is completely generic.

As a drawback, communication is only possible with platforms supporting the required protocols. We outline the basic architecture in the following section and elaborate on the protocol issue in Section 3.2.3.

3.2.1 Basic Architecture

The basic architecture for X-Window–based image transmission is illustrated in Figure 3.1. The application is executed on the render server (utilizing any remote terminal software) with the environment variable `DISPLAY` being set in a way such that both graphical content and user interface elements are displayed on the interaction server. The interaction server will thus receive two kinds of requests: Straight X calls regarding the creation of non-GLX drawables (usually graphical user interface elements) and GLX calls setting up and managing the rendering context (OpenGL calls are encapsulated in GLX request and thus will be passed to wherever the context has been created (see Section 2.8.2); the stream of OpenGL commands, therefore, must not be considered isolated). To allow for user interaction with the application, we must keep on sending straight X calls to the interaction server. In contrast, exploiting graphics resources on the render server requires the GLX drawable and context to be created on this very host. This, in turn, requires the interception and redirection of any call regarding context creation and context management. Naturally, this redirection of GLX requests will not go unobserved on the interaction server: where there visualization used to be, a gap will now be seen that needs to be closed. Similarly, on the render server, no window has been created for attaching the rendering context to since windows are created using X calls rather than GLX; thus, another window must be created on the render server matching the size and visual class of the window on the interaction server that was intended for rendering in the first place. Upon

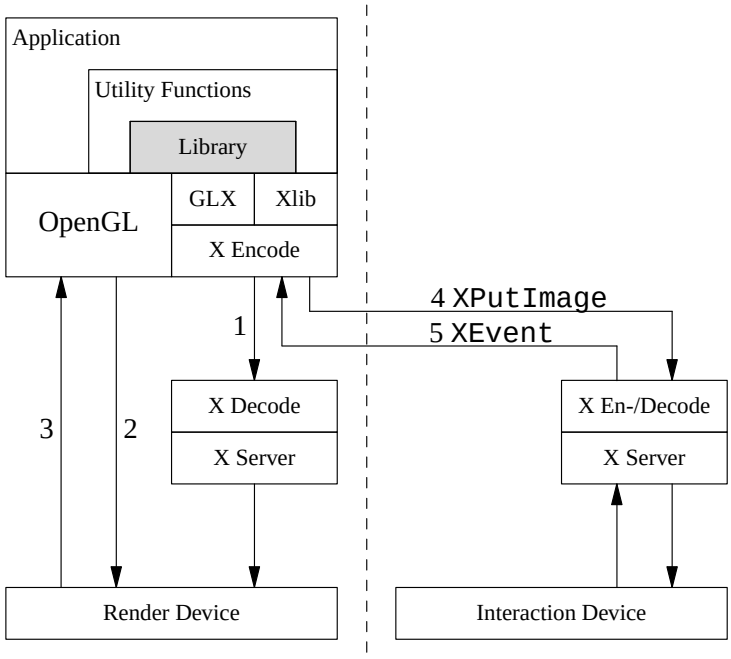

Figure 3.1 Basic system architecture of the remote visualization solution. Interfaces between boxes should be read as *"can/does call functions of ..."*. (1) The application issues a GLX request which is sent to the render server. (2) The application issues OpenGL calls which are handled by the render device. (3) The library reads the contents of the framebuffer and (4) sends them to the interaction server using `XPutImage` requests. (5) `XEvent`s are sent from the interaction server to the application.

swapping buffers, a bitmap of the visualization is read from the framebuffer, encapsulated in an `XImage` structure, and transferred to the interaction server by issuing an `XPutImage(3X11)` call. Since user interactions are not disturbed by redirecting GLX (and, accordingly, OpenGL calls), events will be passed unaffectedly between the hosts involved and there will be no observable effects from splitting of the request stream (besides some inevitable additional lag caused by the network transmission).

3.2.2 Implementation

Attaining the described architecture requires customization of both X and GLX functions. The functions can be classified into set-up functions preparing the rendering context and trigger functions indicating whether graphical content has to be transferred from the render server to the interaction server. We will discuss them in this order.

Context-Related Functions

Obviously, the better part of context-related functions are GLX calls and we have shown that all of them—more than 50 in total—need to be customized to guarantee proper context set-up and management. Doing this manually is tedious and error-prone. Fortunately, however, the majority of GLX functions expects the display as a first argument. As was shown in Section 2.8.1, the display identifies the X server connection; thus, overwriting the display variable to point to the render server is necessary in either case. Similarly, it shows that most GLX functions only require a uniform customization regarding drawable identifiers and rendering contexts to cope with applications with several rendering contexts like multi-view applications. These functions can thus be customized automatically by organizing the mapping of X drawables to GLX drawables with a hashtable. A number of X and GLX functions, however, need to be adjusted manually.

An application initiates a connection to an X server by calling the function `XOpenDisplay(3X11)`. Since we require two server connections—one to the render server and one to the interaction server—we customize this function to open a second connection to the render server and to return the requested display handle. Analogously, `XCloseDisplay(3X11)` is customized to terminate the connection to the render server when the connection to the interaction server is being closed.

A graphical application will usually proceed by specifying the color handling of the GLX drawable. These specifications are commonly referred to as a *visual* and the process accordingly is known as *choosing a visual*. A visual is chosen by calling `glXChooseVisual(3G)`. This function must be customized to, first, redirect the call to the render server in order to obtain a visual valid on the machine where the rendering is going to take place (as it must be done for all GLX functions) and, second, to select a visual on the interaction server suitable for displaying the rendered image. Both visuals must match as close as possible. In our case, the term “match” is defined with regard to the visual class (i.e. the way in which pixel values are translated into visible colors) and the color depth (i.e. the number of bitplanes used for representing the color). While in practice using current hardware this matching usually will succeed, it is not destined to do so. Thus, when the matching fails this is indicated to the application. In general, a well-written application will be prepared for this case and accordingly be able to cope with the situation by adjusting the visual characteristics.

If, on the other hand, a match is found, the visual of the interaction server is returned and the attribute set corresponding to the render server saved for the creation of the rendering context. The latter is required for rendering into windows or pixmaps and accomplished by calling `glXCreateContext(3G)`. It cannot be guaranteed that calls to `glXChooseVisual` and `glXCreateContext` corre-

spond; thus, saving the attribute list and assuming that a `glXCreateContext` call corresponds to the preceding `glXChooseVisual` call is not safe.

A better solution would be to insert the attributes in a hashtable with the return values of `glXChooseVisual` (visual information with respect to the interaction server) as keys. In `glXCreateContext` we can then use the visual information passed as an argument to look up the attribute list. This, however, is still not optimal—an application might not use `glXChooseVisual` at all (albeit this case did not show up for applications tested in this work). In practice, saving the attribute set proves reliable so the hashtable solution was not implemented.

Upon the creation of the rendering context by the client a valid rendering context on the render server has to be created. Whenever possible, the rendering should be done into an off-screen pixel buffer (*pbuffer*) since this kind of drawable does not suffer from obstruction by other windows. Unfortunately, a pbuffer must be requested during context creation while the actual pbuffer dimensions are not known until the GLX context is attached to a drawable using `glXMakeCurrent(3G)`. As a result, it is unknown if a pbuffer of the requested size can be created or if we have to resort to a standard GLX window. This presents a classical hen-and-egg problem: The call to `glXCreateContext` cannot be completed until `glXMakeCurrent` has been called, and `glXMakeCurrent`, in turn, cannot be called unless a context is available which is expected as an argument. Two solutions come to mind.

The first solution is to return an invalid context and to defer the context creation. The problem with this solution is that applications might be tempted to obtain context information (using `glXQueryContext`) using the handle returned by `glXCreateContext`. Of course, `glXQueryContext(3G)`, too, could be intercepted but this would not help at all in supplying valid context information and, thus, might disrupt program behavior. As a second solution, we might assume that the pbuffer creation will succeed and resort to standard GLX windows if the assumption fails.

We implemented the latter solution and duplicate the drawable when `glXMakeCurrent` is called. The duplication involves creating a pbuffer or a window and the creation of an `XImage` structure for transferring the graphics content from the render server to the interaction server. If no pbuffers are supported by the render server, we need to assure that specifications regarding the window size are not reinterpreted by the window manager (which, as discussed in Section 2.8.1, is free to interpret the size as a hint only). Therefore, window manager control is turned off for the respective window.

For static windows, i.e. windows of constant dimensions, this concludes the discussion of functions requiring customization. When resizing application windows is allowed, the OpenGL function `glViewport(3G)` additionally must be overridden to adjust the size of the pbuffer or the GLX window, respectively, to

account for the new dimensions of the corresponding X drawable. Again, making sure the window manager does not interfere the resizing is essential.

Trigger Functions

As argued in Section 2.8.2, double buffering is ubiquitously used to prevent the user from seeing unfinished images. Thus, the corresponding function provoking buffer swaps, `glXSwapBuffers(3G)`, is a natural choice for determining whether the rendering is finished and whether the final image must be transmitted to the interaction server. Accordingly, when `glXSwapBuffers` is called the framebuffer content of the current GLX drawable on the render server is read into the corresponding image structure with `glReadPixels(3G)` and transmitted with an `XPutImage` call. Since the image data has to be copied from client memory to server memory this incurs a high performance penalty. If available, the MIT X Shared Memory extension and `XShmPutImage` are used instead. This extension, however, cannot be used if client and server are running on different hosts as as it is normally the case in a remote visualization environment. Nevertheless, in Section 3.2.3 we will show that this functionality is still useful.

In practice, customizing `glXSwapBuffers` works well. An application, however, is not required to use double buffering and nothing at all will be seen if an application defines the termination of the rendering process with non-GLX calls. Two OpenGL calls designed for this purpose are `glFlush(3G)` and `glFinish(3G)`. Obviously, customizing these functions by invoking the image transmission code already used for `glXSwapBuffers` is trivial. However, it is also performance-impairing for double-buffered applications since buffer swaps implicitly flush all command streams, so care should be taken.

We know that an application creates a context using `glXCreateContext`. As was explained above, the context creation requires knowledge of the attribute set passed to `glXChooseVisual`. Thus, when the context is created, we can establish a mapping from a given context to the corresponding attribute set by, e.g., inserting the attribute set into a hashtable accessed using the returned context (it is actually a pointer, i.e. an integer value) as a key. When the drawable is duplicated in `glXMakeCurrent`, the function is called with the context returned by `glXCreateContext`. Since the GLX drawable is also known, the attribute set—accessed via the context handle—can now be associated with a GLX drawable. Finally, when `glFlush`, `glFinish`, or `glXSwapBuffers` is called, we can determine the current GLX drawable (`glXGetCurrentDrawable(3G)`) and access the attributes of the associated context. By parsing the set for the flag `GLX_DOUBLEBUFFER`, the buffering mode can be determined and the requested operation involving the image transmission performed or discarded.

Although this solution works in theory, it has not been implemented for this

work, the reason being that the context handle of type `GLXContext` is intentionally declared as an opaque type. Exploiting the knowledge about it being a pointer (and thus being a suitable key for accessing hashtable entries) therefore is bad programming style and highly error-prone.

In [95] detailed performance measurements for the given basic remote visualization system are given for an NVIDIA GeForce 2 graphics card (framebuffer read-back rates are strongly hardware-dependent). Naturally, the framerate drop is drastic for applications with high communication overhead, i.e. applications with extremely high framerates (hundreds to thousands of frames per second). In this case, the network becomes the limiting factor—a hardware problem rather than a software issue. For CPU-bound applications, however, which are dominant in numerical flow visualization and scientific visualization in general, we measured a framerate drop by about 30-40 percent. Using standard 100 Mbps local networks, this performance loss still allows for interactive work with many real-world applications. Furthermore, the GUI of the application remains completely functionally intact. And since advanced toolkits like Qt and GLUT are just wrappers for the low-level GLX interface, the presented solution will also work with any of these toolkits, albeit they have not been addressed explicity in the above discussion. The system is thus truly generic.

3.2.3 Optimizations

The basic remote visualization solution suffers from high image transfer costs when using large viewport sizes (the viewport determines what part of the window can be affected by the drawing commands; the viewport size usually matches the size of the drawable). For high-framerate applications this merely results in a sharp framerate drop. Nevertheless, the application might still be usable interactively. In contrast, applications running at barely interactive framerates will be rendered unusable when executed remotely. This motivates the need for optimized data transfer.

In the basic solution the image transfer is accomplished by means of a single `XPutImage` call. This means, if data compression is to be used for accelerating the data transmission, we must rely on what is provided by the X Window system. Unfortunately, image compression is not part of core X. The situation can be alleviated to some extent by using *Low-Bandwidth X* (LBX) [25]. LBX is an X extension designed to improve display and response times in slow networks by compressing the byte stream of the X protocol. LBX is, thus, a general stream compressor rather than a specialized image compression solution. The idea behind LBX is to install a proxy X server which clients connect to. The actual proxy server is `lbxproxy(1)` which caches and compresses the requests and forwards them to the X server the clients want to communicate with. The receiving X server,

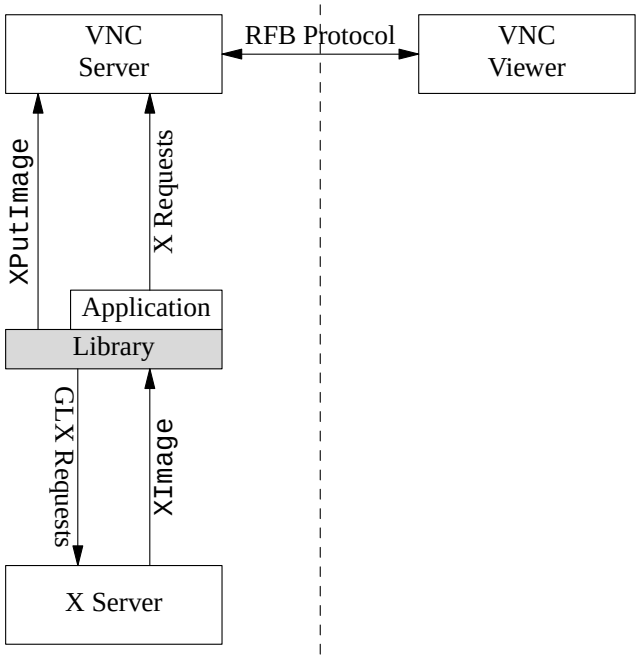

Figure 3.2 System architecture of the basic remote visualization system used in conjunction with VNC.

in turn, recognizes the compressed byte stream and decompresses it. This process is completely transparent to the client and is thus easily integrated into the remote visualization system. Measurements, however, show that although network traffic is reduced when utilizing LBX, no performance gains can be observed in a local area network (LAN) environment.

As an alternative to LBX, the *Virtual Network Computing* (VNC) [73] system can be used. VNC is a well-established, freely available remote desktop tool, i.e. it allows for accessing graphical user interfaces of hosts connected to the local machine by some network. VNC includes both a server and a client component, each of them being available for any relevant platform. The server part is platform dependent and on Unix-based systems is based upon a standard X server. The client role can be played by specialized viewers but also by any Java-enabled web browser. A common protocol—the *Remote Frame Buffer* (RFB) protocol—ensures inter-platform communication.

When using VNC, the complete desktop is transmitted as an image to the client which can run applications on the server and interact with them as if working locally. The image transmission is triggered by a watchdog process that detects framebuffer changes on a per-pixel basis, extracts rectangular areas that need to be updated and finally transmits the image using various specialized, lossless image

compression algorithms. Since the server component, `vncserver(1)`, is based on a standard X server, no extensions are supported. This means, GLX—which is also implemented as an extension—is not supported either. Therefore, VNC as-is does not qualify for remote visualization. On the other hand, we can take advantage of this very fact that `vncserver` is based on a standard X server since the basic remote visualization library was designed to communicate with any two X servers. Thus, to use the remote visualization system with VNC, all that has to be done is to specify the VNC server as destination for GUI elements and graphical content instead of the interaction server, or, put another way, the VNC server becomes the new interaction server (Figure 3.2).

For the standard *gears* benchmark, using VNC for image transmission results in a framerate gain of almost a factor of two. In general, significant speed-ups can be expected for an application whose graphics content includes large areas of uniform color since VNC's *hextile compression* (like any other image compression algorithm) performs exceptionally well in these regions. For a volume rendering application exhibiting little graphical content of uniform color, no speed-up can be observed. In fact, the overhead introduced by VNC even results in a marginally lower framerate. The average application, however, can be expected to reside between these two extrema making the utilization of VNC generally profitable.

Using VNC not only has performance benefits. One of the main limitations of the basic remote visualization system is the requirement to run X servers on both the machine hosting the render server and the machine hosting the interaction server. With VNC, this requirement still holds for the render side. However, VNC supports numerous client platforms and also includes a platform-independent Java-applet client. Thus, using VNC, client-side platform-independence and inter-platform operability come for free—leveraging well-established remote-access software like VNC is, therefore, also economical from a software development point of view. Figure 3.3 shows two application examples of the remote visualization system. The user interfaces are fully functional and hide the fact that the applications are running remotely. Both applications are served by a Linux host and displayed on a MS Windows platform.

3.3 Dedicated Channel Image Transmission

The optimal remote visualization system has several desired properties. First, the system should be easy to use: The user should be presented the identical user interface and should not have to specify more than the name of the interaction server—the minimum information. Second, the solution should be generic, i.e. it should not require modifications of the applications since eventually this will make the difference between being widely accepted or rejected by the users. Third, it should

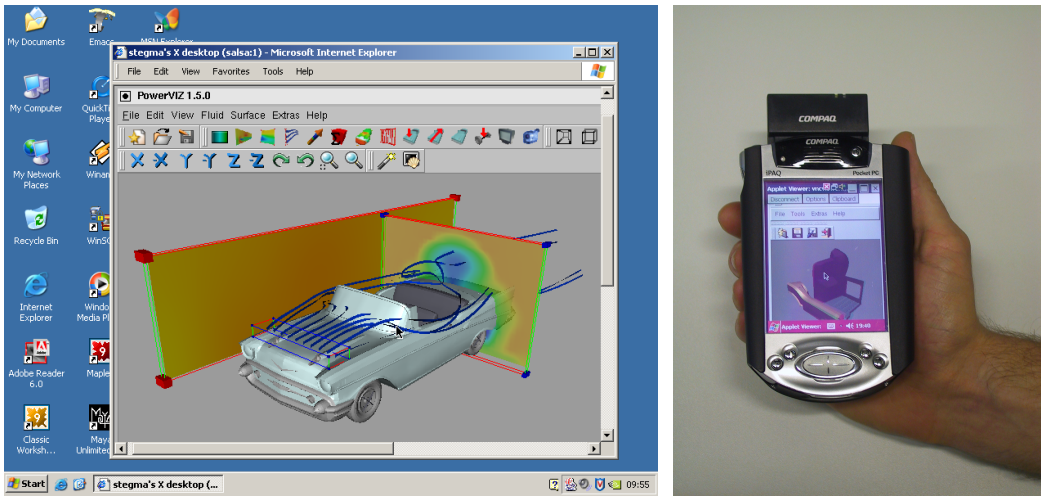

Figure 3.3 Remote visualization applications. Left: Flow visualization software operated in a web browser. Right: Post-processor for crash simulations operated on handheld computer. A Linux-based render server is used in both cases.

be platform-independent, i.e. working with a Unix application using a Microsoft Windows machine should be possible. Finally, the performance should allow for interactive work. Rating the basic remote visualization solution—considering both X display redirection and VNC for image transmission—according to these criteria gives the following table:

	X Windows	VNC
Usability	++	++
Genericness	++	++
Platform-independence	−	+
Performance	−	○

There is no doubt that the basic remote visualization system is easy to use and generic. Furthermore, as shown the requirement to run an X server on the interaction side can be abolished in combination with VNC. Using VNC, however, was not motivated by the demand for platform independence but rather by the lack of image compression functionality when using straight X functionality for image transmission. Still, the use of VNC is not optimal since transmitting the entire desktop and the general design create unnecessary overhead. VNC, thus, does not completely solve the performance issue.

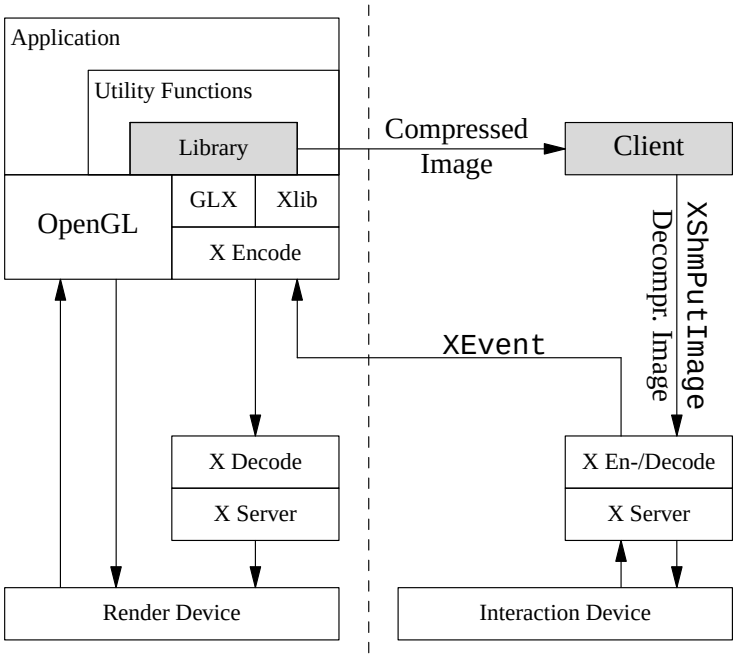

Figure 3.4 Revised architecture of the remote visualization solution allowing for various optimizations.

3.3.1 Revised System Architecture

Ideally, the remote visualization system would allow for an integration of specialized image compression algorithms. In its current form, however, incorporating new compression algorithms requires modifications of the X server, i.e. either of LBX or the VNC server. Alternatively, the server-only design can be dropped and the architecture revised. In this work, the latter approach was followed. The novel architecture and various optimizations were first presented in cooperation with J. Diepstraten and M. Weiler [91].

The required modifications to the existing design are minor: a dedicated data channel for transmitting image data is set up and the data received by a special client application running on the interaction server. The client decompresses the data stream and displays the image again using `XPutImage`. Since `XPutImage` is now called on the interaction server, shared memory extensions can be employed for efficiently transmitting the image data to the X server via `XShmPutImage`. The revised architecture is illustrated in Figure 3.4.

The original architecture comprises only a single component (the pre-load library) and is easy to use since basically only two environment variables had to be adjusted: the variable `DISPLAY` to point to the interaction server and the variable `LD_PRELOAD` to point to the remote visualization library. Since usability

contributes to the acceptance of the remote visualization solution, ease of use is a property that should not be sacrificed in the transition to the two-component design. When examining the invocation of the one-component solution (Bourne-shell syntax):

```
$ export DISPLAY=interaction_server:0.0
$ LD_PRELOAD=$PWD/librv.so.0 opengl_app
```

it is seen that the information necessary for addressing the image data can be extracted from the value of the `DISPLAY` environment variable. Thus, at least the server can be invoked as before without having to specify any additional arguments. The client, on the other hand, requires both the image data and information about the destination window. The former obviously is available to the client (it was designed for receiving this very data), the latter is information already required for the original solution. Thus, this information can be passed to the client as header data along with the image data. The client application, therefore, can be run without any command line arguments and this, in turn, means that the new design allows for the integration of optimizations while being as user friendly as the basic remote visualization system. Unfortunately, the new design no longer can be combined with VNC since in this case we have no means for identifying the destination window (it does not exist anymore but is only a subregion of a larger window representing the entire desktop) and, accordingly, it sacrifices platform independence in favor of reduced data rates and latencies.

3.3.2 Data Rate Reduction

For this work two data reduction methods were integrated. First, data reduction by incorporating image compression algorithms and, second, special motion handling for extremely low-bandwidth networks reducing the viewport during user interactions.

Image Compression

Image compression algorithms exploit redundancies in the input signal and attain high data reduction rates with introducing only minor visual artifacts (lossy compression) or none at all (lossless compression). Integrating compression into the revised architecture is trivial: after having read the framebuffer data, the server passes the data to the compression routine and transmits the returned data. The client, in turn, passes the data received on the network connection to a decompression routine and uses the returned data for constructing an `XImage` structure. Thus, the more relevant question is which compression algorithms should be integrated. Three criteria can be defined. First, the algorithm should allow for fast

Table 3.1 Framerates of *gears* for different scenarios. Left: 802.11b WLAN, 240×320 viewport (Compaq iPAQ display size), right: Fast Ethernet, 512×512 viewport. CCC_LZO denotes LZO-compression of CCC-compressed image data.

	WLAN			Ethernet		
Compression	Ratio	PSNR	FPS	Ratio	PSNR	FPS
none	–	–	0.6	–	–	10.2
LZO	17	–	9.0	33	–	170.0
ZLIB	40	–	9.4	79	–	72.0
CCC	8	37	7.5	8	42	138.0
CCC_LZO	52	37	10.0	89	42	138.4
BTPC	30	31	3.5	58	34	20.0

compression and decompression to keep latencies low. Second, the algorithm should result in high image quality and it should be able to preserve pixel-wide lines. Third, the compression ratios should be high (or even adjustable) in order to be usable in low-bandwidth networks. In this work, five algorithms were integrated and evaluated.

The class of general-purpose compression algorithms is represented by the ZLIB library and LZO. ZLIB [26] is generally known for good performance and high compression ratios. Since it is readily available on most Unix installations it can serve as a fall-back solution. LZO [62] is optimized for speed instead of compression ratio and especially designed for real-time data compression and decompression. Another benefit of LZO is that decompression is computationally simple and, accordingly, very fast—a highly welcome property for a remote visualization system which, by its very definition, assumes that the client (interaction server) has low performance.

For general image compression we integrated *Binary Tree Prediction Coding* (BTPC). BTPC [75] is advertised to be fast and to generally perform moderately well with regard to compression ratios. On average, the algorithm performs better than other image compression for lossy compression of photographs (which medical volume visualizations resemble) and significantly better than JPEG for lossy compression of graphics (which is relevant for engineering applications like numerical flow visualization). For lossless image compression BTPC performs comparable to state-of-the-art JPEG-LS and PNG. In the context of remote visualization, the ability to switch between lossy and lossless compression is advantageous since it allows for an adaptation to network bandwidths and client/server performance. Another general image compression algorithm integrated is *Color Cell Compression* (CCC). The main benefit of CCC [9] is a fixed compression

ratio (1:8) and a fixed latency which in a remote visualization system can be used for framerate guarantees. In addition, as with LZO decompressing CCC-encoded data is computationally cheap and thus suitable for low-performance clients. CCC is being used in a slightly modified version in SGI's Vizserver. We will elaborate on the CCC algorithm in Section 3.3.3.

All algorithms were evaluated with both low-bandwidth wireless networks (11 Mbps) and high-speed local networks (100 Mbps) and with regards to the resulting image quality (Table 3.1). For the latter we computed the *peak-signal-to-noise ratio* (PSNR) commonly used for evaluating compression algorithms and defined by the ratio of maximum pixel values to the mean squared error on a logarithmic decibel scale [63]. Values greater than 30 usually indicate good image quality. The measurements performed using the standard *gears* benchmark application show that—despite being lossless—LZO compression is a suitable choice in both low-bandwidth and high-bandwidth networks and that it is generally superior to the other compression algorithms taking part in the evaluation. Compared to image transmission using straight X functionality a framerate gain of more than one order of magnitude is obtained with the revised remote visualization architecture and LZO compression. Compared to VNC-based image transmission (Section 3.2.3) which only resulted in a twofold framerate, this is a significant improvement outweighing the inherent loss of client-side platform-independence. In conclusion, we found that applications which are highly non-interactive using standard X transmission or VNC exhibit interactive framerates when run with the revised library and custom compression. And this we found is also true for wireless networks with measured peak transmission rates of less than 200 KBps. Thus, the proposed architecture also qualifies as a practical mobile remote visualization solution.

Motion Handling

Compression is just one means for reducing the data rate, another is to take advantage of the fact that the human eye is unable to recognize fine details on rapidly moving structures. Thus, detailed data can be presented where required and less detailed data where possible. In the context of computer-generated visualization, moving structures are the result of user interactions and animations. By monitoring the trigger functions as seen in Section 3.2.2, the average number of images per time unit can be determined. Based on this information, a heuristic can be used to detect user interactions and appropriately react to this situation. The question about what "appropriate" means, i.e. what reaction promises the highest speed-up, can be answered by analyzing the major phases of remote visualization and the most expensive operations involved with each phase.

The time needed for rendering the scene depends on the scene complexity and

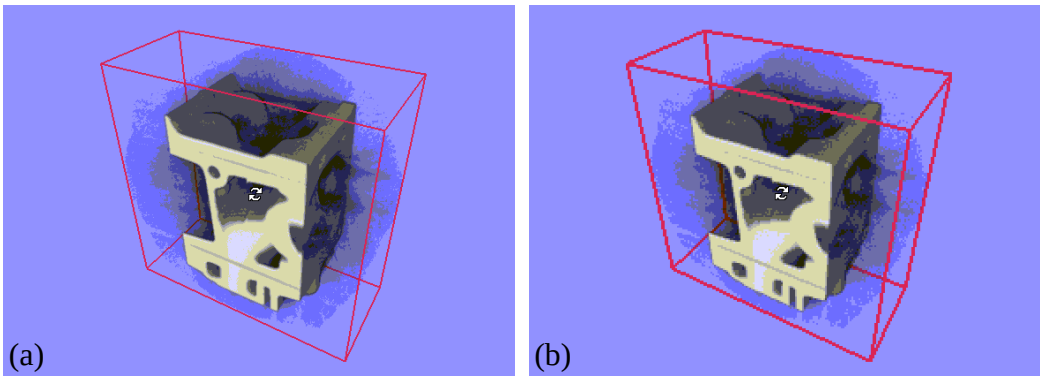

Figure 3.5 Motion handling by the example of a volume renderer. Downsampling effects are clearly visible at the bounding box but almost invisible in the volume. Note that the mouse cursor in the center is shown in high resolution in both images since it is not part of the rendered image.

the viewport size. The next step, reading the framebuffer, depends on the amount of data that has to be read. This also applies to the compression phase—the more data needs to be compressed, the longer it will take. Finally, this also applies to the data transmission; thus, the viewport size has an impact on all phases and reducing the viewport size will result in a performance gain on several levels.

In our system, the scene is, therefore, optionally rendered with half the viewport size in both dimensions (the factor of two is chosen arbitrarily) during user interactions. As before, the smaller image is read from the framebuffer, compressed, and transmitted. The original image size is restored on the client side by duplicating pixels. The framerate increase gained from motion handling is more than 50 percent but it is less than expected—one might anticipate a framerate increase by a factor of four—since the client has both to rescale and mirror the image data for the `XImage` structure in software. In addition, whether downsampling yields an acceptable image quality strongly depends on the application. Considering, e.g., the volume rendering application shown in Figure 3.5, it becomes apparent that while some graphical content is almost unaffected, the quality of other content will significantly suffer from downsampling. Thus, we recommend motion handling only for networks of extremely low bandwidth.

3.3.3 Latency Reduction

Motion handling reduces the amount of raw data and thus speeds up consecutive phases, thereby decreasing the time elapsing between user interactions and the moment the corresponding image is displayed on the interaction server, i.e. the *latency*. Motion handling is thus a hybrid approach for both data reduction and

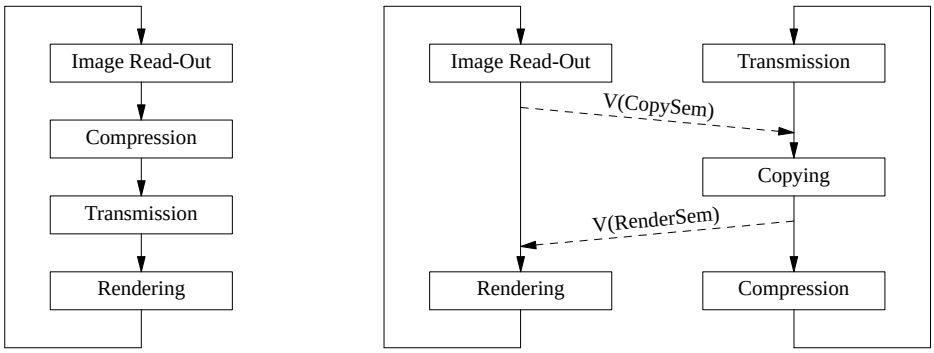

Figure 3.6 Parallelization of the render server work (left) into GPU-dominated work and CPU-dominated work in the revised remote visualization architecture.

latency reduction. Two other latency reduction methods were implemented for this work: parallel processing and GPU-based image compression.

Parallelism

Considering the individual steps of the remote visualization system on the render server, it can be observed that there is both GPU-intensive and CPU-intensive work. Four major phases can be identified: rendering, framebuffer read-out, compression, and transmission. Assuming the use of display lists for rendering (display lists are server-side caches for OpenGL commands) and *Direct Memory Access* (DMA, allows for reading and writing system memory without interrupting the CPU) operations for reading the framebuffer content, these two operations can be classified as GPU-dominated work. Accordingly, compressing the image and transmitting the data are CPU-dominated work. Any partitioning like this is irrelevant as long as the operations are performed sequentially. On the other hand, both the GPU and the CPU are autonomous processors able to perform their work independently—assuming, of course, that none has allocated resources required by the other one. For example, if the CPU is blocked by some network routines waiting for acknowledgments of the receiving side, the scene cannot be redrawn although a single function call would suffice to initiate the display of some geometry stored in a display list or vertex buffer. By introducing a suitable parallel architecture as depicted in Figure 3.6 these stalls can be reduced.

In this work, a parallel architecture was implemented with the POSIX *pthreads* library and semaphores. The two threads running in parallel are depicted by two cycles in Figure 3.6. The GPU-thread continuously renders the scene and transfers the image data to main memory. In order to be sure that the next view may be rendered and made available to the CPU for compression, the thread needs to make sure no previous image data is overwritten. The CPU-thread, on the other

hand, needs to know when new image data is available. The latter is indicated by issuing a pass-operation (traditionally indicated by `V(semaphore)`) on the copy-semaphore, the former by a pass-operation on the render semaphore. Measurements (see [91]) show that by this approach a framerate increase by almost 50 percent can be attained.

GPU-Based Image Compression

Optimizing the software compression libraries used in this work with regard to compression time clearly is capable of reducing latency. The framebuffer readout time and the data rate, however, stay constant. On the other hand, if the image compression is accomplished directly on the GPU, not only latency profits from this optimization but also framebuffer reads since less pixel data would have to be transferred from GPU-memory to main memory. As was shown in Section 2.6.2, a GPU can be considered a multi-processor; thus, similarly to the approach presented in the previous section, GPU-based image compression is also a form of exploiting parallelism, albeit in a less traditional form.

In this work, image compression in hardware has been implemented for the CCC algorithm which, as shown in Table 3.1, is second only to LZO in environments with fast networks. The choice of CCC is easily motivated by inspecting the algorithm in detail. We assume the input image to be 24 bit RGB. Furthermore, if the number of pixels in any of the two dimensions of the image does not evenly divide by four, we assume that the image is padded by inserting dummy pixels. The CCC algorithm proceeds by first splitting the image into *cells* of 4×4 pixels. Each cell is processed independently of the other cells.

The first cell operation is to determine the mean luminance $\mathcal{L}_{\text{mean}}$ which is computed by averaging the luminance values of the individual pixels. As usual, luminance is computed with the well-known luminance equation $\mathcal{L} = 0.3\mathrm{R} + 0.59\mathrm{G}+0.11\mathrm{B}$ [23], where R, G, and B denote the red, green, and blue components of the color, respectively. Of course, the luminance value of a cell pixel is either less than or equal to the mean luminance $\mathcal{L}_{\text{mean}}$ or greater than $\mathcal{L}_{\text{mean}}$. This fact is used in the CCC algorithm to classify the pixels and to create two pixel groups $\mathcal{P}_1$ and $\mathcal{P}_2$ having mean colors $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, defined by the arithmetic mean of the corresponding pixel colors. The colors $\mathcal{C}_1$ and $\mathcal{C}_2$ furthermore are quantized to two byte by just storing the 5, 6, and 5 most significant bits of the red, green, and blue components, respectively. Obviously, since a cell comprises a total of 16 pixels, an equivalent formulation for $\mathcal{P}_1$ and $\mathcal{P}_2$ is a 16 bit bitmask having a bit set where the corresponding pixel's luminance is greater than $\mathcal{L}_{\text{mean}}$. The basic idea of color cell compression now is to encode an entire cell with only six bytes: the two-byte bitmask and the quantized colors $\mathcal{C}_1$ and $\mathcal{C}_2$ also requiring two bytes each. Given that the original cell requires $16\times3 = 48$ bytes,

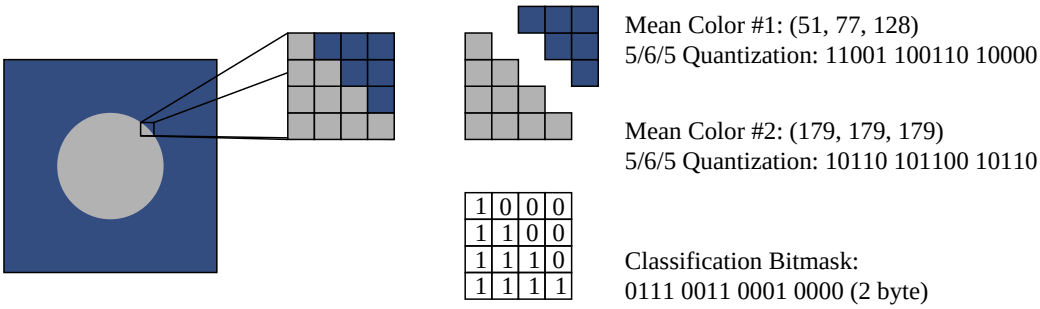

Figure 3.7 Color Cell Compression. The input image is split into 4×4 tiles the pixels of whose are classified with regard to a luminance threshold. For each class a mean color is computed and quantized to two byte. For decompression a pixel's value is set to either one of the two colors based on the classification bitmask.

this results in a compression ratio of 1:8—regardless of the data being encoded. For decompressing the image, a pixel is assigned color $\mathcal{C}_2$ if the corresponding bit in the bitmask is set, otherwise color $\mathcal{C}_1$ (Figure 3.7).

Color cell compression has two remarkable properties: First, all cells are processed independently and, second, in contrast to run-length–based algorithms the amount of both the input and the output data is fixed. Thus, the CCC algorithm ideally matches the criteria defined in Section 2.6.2 that an algorithm must possess in order to be suitable for an implementation in graphics hardware.

What follows is a description of the GPU implementation for the NVIDIA GeForceFX graphics hardware but the implementation was also adapted to the ATI Radeon 9700. The graphics API is OpenGL. Instead of transferring the framebuffer content to main memory for compression, the GPU implementation first copies the rendered image to a texture using `glCopyTexImage2D(3G)`. Next, a quadrilateral is rendered the size one fourth of the original image in both dimensions, i.e. one fragment of the drawn rectangle maps to one cell of the original image. A fragment program is bound that performs this mapping by issuing 16 texture look-ups to fetch the pixels corresponding to the respective cell. Once the input data is available, computing the mean colors and the bitmask can then be performed almost exactly as it would be done in a software implementation by leveraging a high-level shading language (in this case Cg). The relevant source code snippets are shown in Figure 3.8. Note how vector operations can be efficiently used to simultaneously compute an entire row of four bitmask entries.

For computing the colors $\mathcal{C}_1$ and $\mathcal{C}_2$ the code for computing the bitmask is reused to a large extent since basically the only difference is to sum up the two cell colors and the number of colors in one class. The mean colors can then be computed by dividing the two color sums by $|\mathcal{P}_1|$ and $|\mathcal{P}_2|$ represented by `(16 - count)` and `count`, respectively.

```
// Compute mean luminance
for (float j = 0; j < 4; j++) {
    for (float i = 0; i < 4; i++) {
        colArray[i][j] = texRECT(sourceImage, tempPos + float2(i, j)).rgb;
        meanCol += colArray[i][j];
    }

    lumArray[j] = float4(dot(colArray[0][j], float3(0.3, 0.59, 0.11)),
                         dot(colArray[1][j], float3(0.3, 0.59, 0.11)),
                         dot(colArray[2][j], float3(0.3, 0.59, 0.11)),
                         dot(colArray[3][j], float3(0.3, 0.59, 0.11)));
}
meanLum = dot(float3(0.3, 0.59, 0.11), meanCol)/16.0;

// Compute the bitmask
for (float j = 3; j >= 0; j--) {
    float4 weight = lumArray[j] > meanLum.rrrr;

    if (j > 1) {
        upperBitMask = 16 * upperBitMask + dot(weight, float4(1, 2, 4, 8));
    } else {
        lowerBitMask = 16 * lowerBitMask + dot(weight, float4(1, 2, 4, 8));
    }

    col2 += weight.r * colArray[0][j];
    col2 += weight.g * colArray[1][j];
    col2 += weight.b * colArray[2][j];
    col2 += weight.a * colArray[3][j];

    count += dot(weight, float4(1,1,1,1));
}
col1 = (meanCol - col2)/(16 - count);
col2 = col2/count;
```

Figure 3.8 GPU implementation of *Color Cell Compression* written in Cg.

Each computation—bitmask, $\mathcal{C}_1$, and $\mathcal{C}_2$—is performed in a single rendering pass so the resulting implementation requires three passes. After each pass, the framebuffer content is read to compile the compressed data to be transmitted to the interaction server. A two-pass solution is also possible by writing the bitmask in a first pass and the colors $\mathcal{C}_1$ and $\mathcal{C}_2$ simultaneously in a second. We found that the compression time scales linearly with the number of passes; thus, the three-pass solution is slower than the two-pass solution by a factor of one and a half. In theory, both the colors $\mathcal{C}_1$, and $\mathcal{C}_2$ and the bitmask could also be written in a single pass, storing the colors in the four-byte RGBA vector and the bitmask in the fragment's depth value (which is a floating point value of at least 16 bit). However, the depth value undergoes a series of (non-standardized) transformations before it is made available to the application; thus, passing general data in the depth value is discouraged.

Similarly to GPU-based image compression, decompression can also be im-

plemented on graphics hardware. Depending on the order of the bitmasks and color as they appear in the compressed data stream, the decompression can either be accomplished with standard OpenGL functionality (non-interleaved data, three-pass solution) or again using programmable fragment hardware (interleaved data, two-pass solution). However, demanding advanced graphics features on the interaction server contradicts the assumption of the remote visualization system that the interaction server is *not* equipped with high-end hardware. We will, therefore, skip the discussion of decompression techniques and refer to [91] for details.

Furthermore, comparisons between GPU-based image compression and decompression and highly optimized software implementations of the CCC algorithm show that the performance benefit gained from the use of programmable graphics hardware is negligible. Given that the whole system becomes more complex and no longer runs on arbitrary hardware, image compression on graphics hardware is, therefore, not recommended. Obviously, the idea of compressing the image directly in hardware also conflicts with the idea of parallelization presented previously since it disrupts the balance between GPU-dominated work and CPU-dominated work and thereby prevents the system from working to capacity. At least in the present context, GPU-based image compression thus must be considered as a mere technology demonstration.

3.4 Usability Issues

A remote visualization library should disrupt program behavior as little as possible. The remote visualization solution of this work (in all its presented variations) in general fulfills this criterion since from a user's point of view it is indistinguishable whether an application is used locally or remotely. However, presenting the application as-is may also pose problems in certain scenarios arising from the remote visualization library. As seen in Section 3.2.3 the presented remote visualization system makes it possible to use handheld computers for interacting with applications running on distant hosts. The screen resolution of handheld devices, however, is completely inappropriate for most applications designed to run on workstations. Thus, although usable in theory, in practice most applications are unusable on small-screen devices when run as-is. Another aspect is stereo visualization. Many applications use stereo visualization to enhance their visualizations. These applications are often designed to provide high-quality stereo vision by simultaneously driving several projectors (Section 2.7). Of course, this functionality has to be disabled for remote visualization, thereby taking away functionality.

This section presents solutions for alleviating these problems. The solutions were first presented in cooperation with D. Rose, G. Reina, and D. Weiskopf [77] and D. Rose [97], respectively.

3.4.1 Adaptation of User Interfaces

The most significant difference between PDAs and desktop computers is not performance but screen size. As a result, using applications designed for desktop computers on handheld devices with their original user interfaces usually is impossible—and this is regrettable since great efforts have been invested into developing these applications. Thus, there is a great interest in using desktop applications on handhelds without having to adapt them (or rather their user interfaces) manually. For this work, a solution has been developed that (at least partially) solves this problem in a manner similar to and equally generic as the way taken for solving the problem of generic remote visualization: by pre-loading the toolkit libraries used by the application for building the GUI. That is, in combination, while the remote visualization solution solves the performance issue by running the application on a powerful remote server, the generic adaptation of user interfaces presented in this section solves the screen size issue.

The solution is similar to what has been presented in [11] in that it also uses a client/server approach where the application is running on the server and the client sends and receives commands and results, respectively. In contrast, however, the solution presented in this work actually displays the (adapted) user interface and neither requires custom clients nor scriptable servers. Our solution, therefore, follows a new paradigm: Instead of implementing a toolkit as an API, the toolkit is implemented as an object file redefining the functionality of an existing toolkit API.

User Interface Design

The first step in redesigning the user interface is to decide what the adapted user interface should look like. Of course, the new look-and-feel is strongly influenced by the properties and capabilities of the target device.

When using a handheld computer, the only input device often is a pen used in conjunction with a touch-sensitive screen. Thus, we can assume that no user interactions besides a click, a double-click, and a mouse drag can be generated, i.e. the new user interface must be very simple. And we have to be realistic, too: It will not be possible to generically map a user interface as included in the flow visualization tool shown in Figure 3.3, left, with dozens of icons, menus, sliders, text entry fields, etc. to a representation that fits on a 240×320 pixel screen, a typical screen size of current handheld computers. It is, therefore, better to accept that a completely generic solution must be abandoned in favor of a more specific solution designed for a single toolkit.

Now the question arises which toolkit to customize. The OpenGL graphics API comes in three libraries: core routines for setting the state of the OpenGL ma-

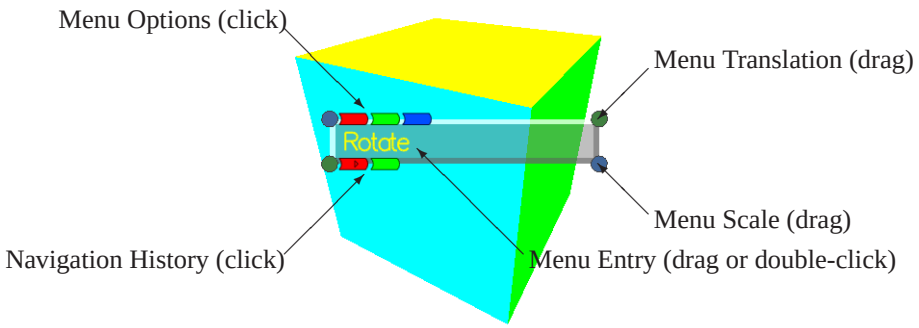

Figure 3.9 Interaction elements of the handheld-suitable cascading menu.

chine, a utility library (GLU) providing utility routines for simplifying common tasks like specifying viewing parameters, and GLUT, a platform– and window-system–independent toolkit for programming user interfaces [38]. The GLUT library is very simple and basically restricted to cascading pop-up menus of arbitrary depths. But, the GLUT library is provided with every OpenGL installation and portable and, therefore, used for numerous applications, especially in academia where functionality is often rated higher than usability. Thus, concentrating on the adaptation of GLUT-based applications is a rational decision since, on the one hand, these applications are widely found and, on the other hand, because GLUT menus are highly space-inefficient and therefore inconvenient to use on small screens.

Of course, screen-space efficiency is of minor importance if the adapted GUI is hard to use. Thus, in order not to neglect this issue, the small-screen user interface design derived for this work is the result of a number of user studies conducted starting from an initial prototype based on the authors' ideas. The prototype was modified in consideration of the study outcomes and re-evaluated, thereby successively improving both its visual design and its interaction techniques. The final design is shown in Figure 3.9.

The adapted user interface follows the metaphor of a cylindrical menu where the menu entries of one level are distributed across the circumference and menu entries can be browsed by rotating the cylinder. This means, when scrolling beyond the last menu entry the first entry is reached and vice versa. The cylinder is visualized as a semi-transparent area labeled with the current menu entry. No curvature information is visualized by shading since smooth transitions cannot be nicely rendered on a small area. The menu area is rendered semi-transparent since it is assumed that there is no possibility for hiding and showing the menu, i.e. the menu is assumed to be visible all the time and it, therefore, must not obscure graphical content.

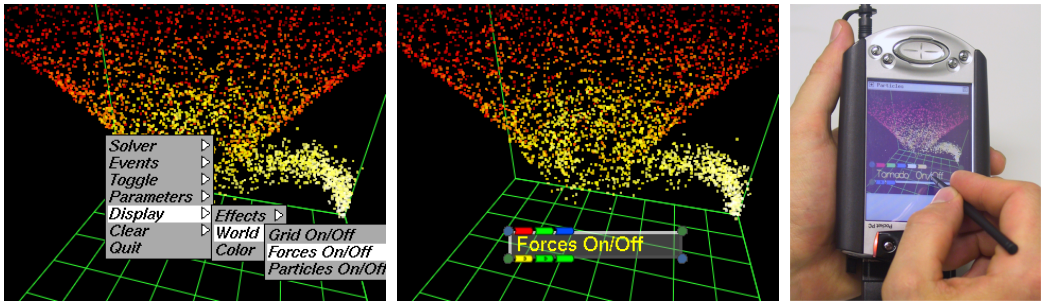

Figure 3.10 Comparison of original GLUT menu and adapted user interface. The application running on a Compaq iPAQ handheld PC is shown on the right.

The menu is browsed by pressing the left mouse button (or rather: pen) inside the entry area and by moving the pointer while keeping the button pressed (sliding the pen across the display). By moving horizontally to the left or to the right, the menu level can be changed. Whenever the current menu entry allows for opening a submenu, this is indicated by an arrow symbol right-aligned in the entry area. In contrast, up/down movements are used for indicating the desire to switch between menu entries on the current menu level. An entry is selected by double-clicking into the entry area.

One problem of a menu showing only a single menu item at a time is loss in efficiency when browsing the menu since—compared to a standard drop-down menu—the user cannot immediately spot the desired menu entry and thus has to inspect the entries one after another. To overcome this limitation, a color encoding of the menu entries is used which visualizes selectable entries on the current menu level by small glyphs at the top of the entry area. The glyphs can be clicked to select an entry. The use of colors helps to memorize entries. A further improvement is a navigation history at the bottom of the entry area. The idea of this history is that by repeatedly selecting an entry, a unique color code is memorized which the user can use to navigate to a desired entry by just clicking the memorized colors successively in the top glyph row. By clicking the glyphs in the lower row the user can also quickly navigate between levels. Furthermore, the possibility to scale and translate the menu is provided. For this purpose, glyphs are positioned criss-cross on the entry area corners so that no functionality is lost when partially moving the menu out of the screen area.

Figure 3.10 shows a real-world example of a 3rd party particle visualization system running remotely and visualized on a handheld computer (right). With the original user interface, the menu covers almost the entire screen space in width, i.e. the screen would be completely cluttered by the original menu (left). In contrast, the adapted menu presents itself in a very space-efficient manner and—due to the use of semi-transparency—it is pretty unobtrusive.

3.4.2 Implementation Details

The task of the pre-load library is to build a new user interface design and behavior based on an existing toolkit without changing the programming interface. For accomplishing this task, an analysis of the user interface structure and the setup of an internal representation for this structure are required.

The structure of a GLUT-based user interface is specified by calling a set of functions for creating menus, dialogs, window elements, and for registering call-back functions. For gathering structure information, these functions must be customized by the pre-load library using the techniques introduced in Section 2.9. The efforts for this customization clearly depend on the extent of the desired modifications. Minor changes are trivial, major changes (e.g., completely redesigning the Qt toolkit) may quickly become a major task and very time-consuming.

In GLUT, a user interface mainly consists of one or several menus. A menu is created by calling the function `glutCreateMenu(3GLUT)`. This creates a new menu and registers a call-back function which will be automatically invoked with an identifier passed as argument identifying the selected menu entry upon selecting the respective item. Once a menu has been created, entries and submenus are added with `glutAddMenuEntry(3GLUT)` and `glutAddSubMenu(3GLUT)`, respectively. One problem about GLUT menus for the intended application is that the functionality attached to an entry is implicitly invoked by GLUT using the call-back mechanism as soon as the entry is selected. In GLUT, selecting a menu entry comprises several steps: First, the mouse button is pressed to show the menu at the current cursor position. Then, while keeping the mouse button pressed, the mouse cursor is moved to the respective entry. Finally, the button is released to select the current entry. Which mouse button is actually used for the selection is configured by the application developer with the function `glutAttachMenu(3GLUT)`.

In the customized version the user interaction is similar in so far as the mouse cursor is moved to the respective entry while holding the button pressed. Activating the entry upon releasing the button, however, is disadvantageous for small-display devices. The reason is that while there is usually plenty of space for physical mouse movements when using a standard mouse, when using a pen there is often very limited space for movements. As a result, when navigating a deep menu hierarchy it may become necessary to lift and reposition the pen in the display center to continue with the interaction once the display border is reached. Thus, if lifting the pen would be interpreted as selecting an entry, selections could be made unintentionally—a very tedious behavior from a user's point of view. A better approach is to explicity demand the activation of an entry by double-clicking the entry area. Feedback for the selection is then given by short flashing.

Obviously, to attain the desired interaction concept, the implementation re-

quires customizing GLUT mouse functions (`glutMouseFunc(3GLUT)` and `glutMotionFunc(3GLUT)`) in addition to functions provided by GLUT for constructing menus. The customized versions save the original call-back function pointers and replace them by pointers to functions which, upon mouse interaction, determine whether the menu area has been hit or missed. In the former case, the menu display is updated according to the user interaction utilizing the menu structure information compiled during the application start-up. In the latter case, the original call-back function specified by the application is invoked. Thus, functionality typically bound to mouse events occurring on the GLX drawable—rotation, translation, scaling—remain unaffected by the interface adaptation.

To draw the menu we need to make sure that the rendering of the scene is finished (otherwise occlusion might be observed). As already seen in Section 3.2.2 this is indicated by the invocation of a trigger function. Like the remote visualization solution, the library adapting the user interface relies on the function `glXSwapBuffers`. Thus, generic adaptation of user interfaces is only supported for double-buffered applications.

3.5 Integration of Stereo Capabilities

In Section 2.7.1 we presented several commonly used stereo techniques for presenting different views to the left eye and to the right eye. Independent of the stereo technique, however, two frusta must be derived yielding the respective views. That means, for implementing generic stereo functionality, three problems must be solved: first, means must be found to implement the stereo technique non-intrusively; second, we need to find out how to derive a stereoscopic view from a given monoscopic view; and third, we have to find a way to render the scene twice without presenting the image after having rendered the first view.

Finally, although not generally required, providing the user with means for configuring the view is strongly recommended since no settings are equally suitable for all users.

3.5.1 Anaglyph Stereo in OpenGL

Regarding the stereo technique we chose to use red/cyan anaglyphs since the required separation of color channels can be very efficiently implemented with standard OpenGL functionality. In fact, anaglyphs have been chosen before for implementing a generic library for polychromatic passive stereo [41]. In principle, this library would suffice for our purposes. However, the implementation never made it to a stable version and—to simplify matters—restricts itself to the customization of GLUT functionality (but, of course, thereby sacrifices the capability of making

non-GLUT applications stereo-aware). Finally, no means are provided to allow for an interactive configuration of stereo parameters. For this work, therefore, a custom anaglyph stereo library was implemented.

The basic idea for implementing anaglyph stereo using OpenGL is simple: the scene is rendered twice, once for the left eye, once for the right eye. When rendering the left eye the framebuffer is configured such that only the red component is modified. Accordingly, when rendering the right eye the red channel is disabled for writing and only green and blue components can be written. The latter two colors result in cyan when mixed, i.e. the two stereo images can be separated with red/cyan glasses. Enabling and disabling writing of framebuffer color components in OpenGL is done with `glColorMask(3GL)`. The display routine thus looks something like this:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
/* Render the scene for the left eye */
...

glClear(GL_DEPTH_BUFFER_BIT);
glColorMask(GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE);
/* Render the scene for the right eye */
...

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glXSwapBuffers();
```

Dissecting the pseudo-code reveals only one unknown function: `glClear(3G)`. Obviously, when rendering the scene twice the viewing frusta and the framebuffer configurations regarding color masking must differ. In either case, however, all objects must be drawn completely. Thus, hidden surface removal (see Section 2.5) must be performed independently for both rendering passes. Because of this, we need to clear the depth buffer *and* color buffer before drawing the next pair of stereo images but only the depth buffer when drawing the second stereo image (in this case the right-eye image) since otherwise the red component drawn so far would be lost.

3.5.2 Derivation of Stereo Frusta

The exposition given in Section 2.7.2 suggests that stereo parameters are dependent on the viewing frustum defined by the application. Therefore, a necessary prerequisite for deriving stereo frusta is to determine the application-provided frustum first.

There is a variety of methods for doing this. First, we can hope for the application calling `glFrustum(3G)`. This, however, is no required behavior. As an alternative, we can also track modifications of the projection matrix but this is tedious. A better approach is to determine the current projection matrix at the very moment the frustum parameters are needed and to extract the parameters from this matrix, i.e. the positions of the near and far clipping planes, n and f, and the positions of the left/right and bottom/top planes (with respect to the near clipping plane), in the following denoted by l, r, b, and t, respectively. In general, the projection matrix P has the following form [109]:

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix} . \tag{3.1}$$

Thus, assuming m_k denotes the kth element of the projection matrix in column-major order, the frustum parameters are given by:

$$\begin{aligned} n &= \frac{m_{14}}{m_{10}-1}, & f &= \frac{m_{14}}{m_{10}+1}, \\ l &= n\frac{m_8-1}{m_0}, & r &= \frac{2n+m_0 l}{m_0}, \\ b &= n\frac{m_9-1}{m_5}, & t &= \frac{2n+m_5 b}{m_5}. \end{aligned} \tag{3.2}$$

That is, we can assume that P is set by the application by calling `glFrustum(l, r, b, t, n, f)`. The goal now is to compute a pair of asymmetric frusta for the given monoscopic view as, e.g., described in [2].

We first define the zero-parallax. Assuming the frustum size has been chosen to snugly fit the scene, good depth cue is usually obtained by positioning the zero parallax plane at about one third of the object depth, i.e. `zps = n + (f - n)/3` as shown in Figure 3.11. Two values need to be determined next (see Equation 2.14): the eye distance and the post-projection shift making sure points on the zero-parallax remain stationary. For comfortable viewing the eye positions should not be shifted by more than about 3.5 percent of the display width. If we assume a screen-filling scene the display width will approximately correspond to the horizontal extent of the projection plane. A first guess for the horizontal extent of the projection plane, in turn, is $r-l$, the distance between the left and right clipping planes. This guess, however, is only accurate if the projection plane is equal to the near clipping plane. Since we do not want to project onto the near clipping

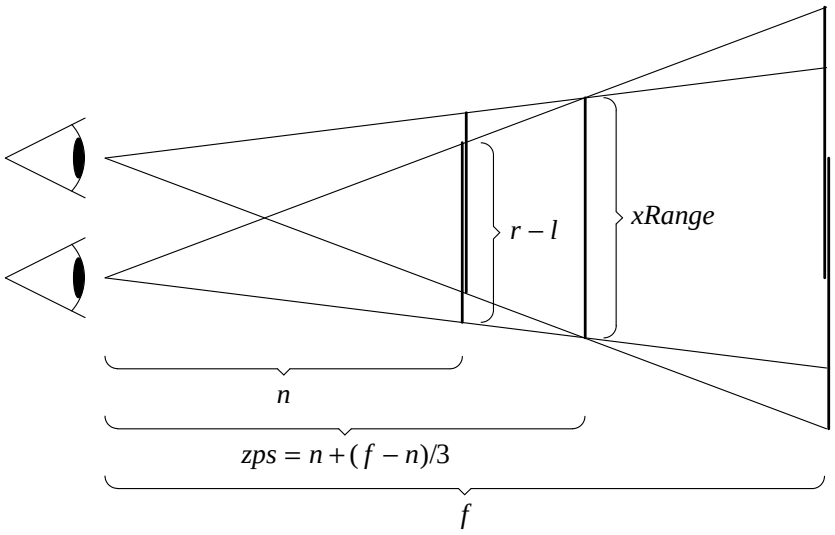

Figure 3.11 Parameters required for computing stereo frusta.

plane but rather onto the zero-parallax plane, we need to refine the initial guess. From congruent triangles it is seen that the horizontal extent of the zero-parallax plane is `xRange = (r - l) * zps/n` and, accordingly, the camera offset `stereoCameraOffset = 0.035 * xRange`. The offset must be positive for the right eye and negative for the left eye, thus, when rendering the left-eye view we negate the stereo camera offset.

The post-projection shift actually must be equal to the negated camera offset. However, since we have to specify the left and right clipping planes with respect to the near clipping plane while the camera offset refers to the zero-parallax plane we again have to determine the correct offset using congruent triangles and obtain `frustumAsymmetry = -stereoCameraOffset * n/zps`. Since we have to shift the scene instead of the camera—OpenGL restricts the camera position to the origin—the scene offset must be positive for the left eye and negative for the right eye. We just negate the camera offset. Translating this exposition to OpenGL code, we obtain:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(l + frustumAsymmetry, r + frustumAsymmetry,
          b, t, n, f);
glTranslatef(-stereoCameraOffset, 0.0, 0.0);
```

where `glTranslatef(3G)` results in a translation by the respective amounts given as arguments.

3.5.3 Redrawing

Calculating stereo frusta is only one necessary component for generating stereo images. Another is the actual rendering of the scene. Ideally, the routine performing the rendering is known and thus can be called once for the left eye and again for the right eye. Unfortunately, however, this routine in general is unknown—it is only known for selected toolkits like GLUT which requires the user to specify the rendering routine using `glutDisplayFunc(3GLUT)`.

One possibility is to use a heuristic. The function `glClear` usually is called right before the rendering starts so we could utilize either the return address of `glClear` (by, e.g., customizing the function and by examining the stack content) or the enclosing function determined with a stack trace based on the return address. Neither of these solutions is guaranteed to work in any case. The following code snippet illustrates this:

```
int main(void) {
    int i = 0;

    /* Open display and create window */
    ...

    /* Clear the drawable */
    glClear(...);

    while (i++ < 10) {
        /* Draw some primitives */
        ...
    }

    return 0;
}
```

When jumping to the return address of `glClear` for re-rendering the scene, the loop is not entered at all for the second view so this solution is incorrect. The enclosing function is no suitable choice either since calling it again results in duplicate X server connections and duplicate windows rather than in both views rendered into the same window.

Alternatively the OpenGL commands issued for rendering a scene could be recorded and played back once for the left eye and another time for the right eye. Display lists (see Section 3.3.3) cannot be used since certain commands are not compiled into the list (see `glGenList(3G)`). This means, we are forced to preload all OpenGL functions and to handle the list compilation and the list play-back on our own—a major task. A comprehensive customization is also required when

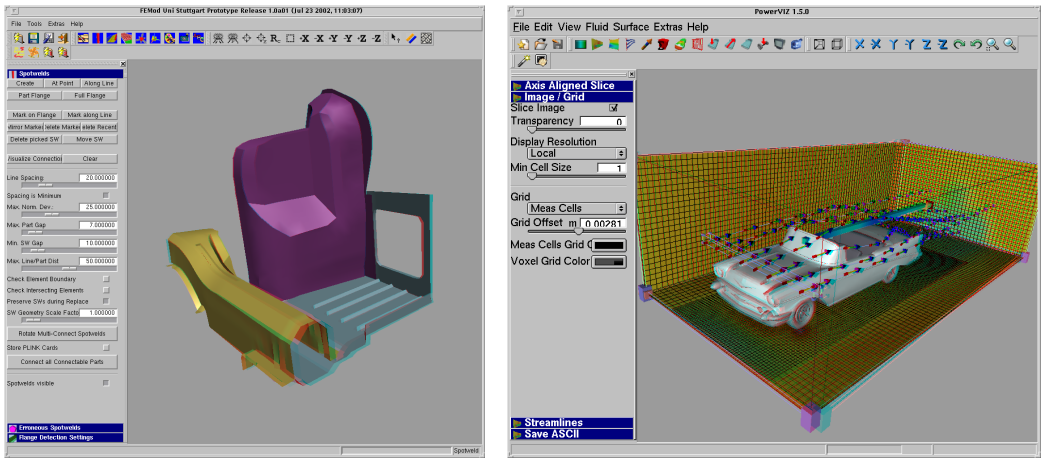

Figure 3.12 Stereo visualization using the generic stereo library. Left: *FEMod*, a commercial pre-processor for crash simulations; right: the commercial flow visualization software *PowerVIZ*.

creating a second GLX context (Section 2.8.2). This solution no longer requires storing OpenGL commands but instead requires frequent context switches which usually is an expensive operation.

A more efficient solution is found by considering what happens if the user drags one window over another. Of course, if the obscured window is exposed, one expects the graphics content to be the same as before. Some X servers provide a mechanism for saving the screen area beneath windows. This mechanism has to be requested explicitly with the so-called *save-under* attribute during window creation. However, saving-under is only a hint since this service is not always available; thus, the application must be able to restore the graphics content itself upon receiving an `Expose` event from the X server.

In this work we exploit this behavior and generate synthetic `Expose` events. This requires pre-loading `glXSwapBuffers` so that we can discard the buffer swap after rendering the first image and set the appropriate projection matrix for the second view. After rendering the second view the buffer swap is accepted and the left-eye view for rendering the first view is restored. Since `glClear` will typically be called between consecutive buffer swaps this function is customized, too, to guarantee the clearing sequence defined in Section 3.5.1.

This event approach not only is advantageous with regard to simplicity and universality but also with regard to performance. Let T denote the time between two buffer swaps, S denote the time needed for the buffer swap itself, and $\text{fps}_{\text{stereo}}$ and fps_{mono} denote the framerates attained when running the application with and without stereo support, respectively. Then the following relations hold (the time

for delivering the event can be neglected):

$$\lim_{\frac{T}{S}\to 0} \frac{\text{fps}_{\text{mono}}}{\text{fps}_{\text{stereo}}} = 1 \ , \qquad \lim_{\frac{T}{S}\to \infty} \frac{\text{fps}_{\text{mono}}}{\text{fps}_{\text{stereo}}} = 2 \ . \tag{3.3}$$

That is, in the worst case the framerate is halved. Considering that for stereo vision two views have to be rendered instead of one, this attests the stereo library near-optimal performance. Two screenshots showing the stereo library applied to two commercial applications are shown in Figure 3.12.

3.5.4 Library Configuration

The above calculations make a number of assumptions regarding the display size with respect to the distance between user and display, the viewport size with respect to the display size, and scene dimensions with regard to frustum extent. Another factor is the user himself since the maximum parallax value considered comfortable varies from person to person as do their eye distances. The user therefore must be given the possibility to manually adjust the default values determined automatically.

Of course, we could pre-load event processing functions and allow the user to interactively adjust the settings by pressing certain keys. Since we do not know what keys are already handled by the application this, however, generally cannot be recommended. As an alternative we could provide a configuration file but restarting the application each time settings have been adjusted quickly becomes tedious. The benefits of the two approaches can be combined by saving the stereo settings in a permanent shared memory segment which basically is a file that can be mapped into memory for reading and writing by non-related processes.

The configuration file initially contains default values. The stereo library maps the file into memory and, thus, can both read and write the settings. For adjusting the values interactively, we provide a configuration program implemented as an X client that also maps the file into memory (thereby making it shared between the stereo library and the configuration program). The configuration program is stand-alone software and so does not interfere with the application. Again, an `Expose` event is sent (now from the configuration tool) to notify the application and, accordingly, the stereo library about modifications of the settings file. Figure 3.13 illustrates the configuration architecture.

3.6 Concluding Remarks

The remote visualization system and the accompanying add-on libraries presented in this chapter constitute a hitherto unknown, unique combination. However, both

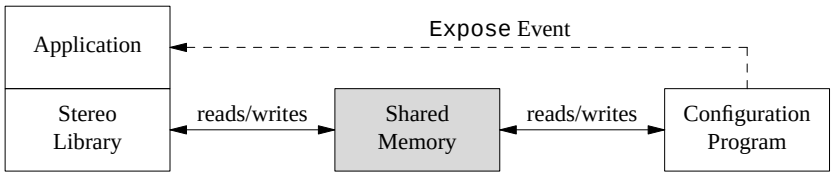

Figure 3.13 Schematic illustration of the stereo configuration architecture.

the GUI adaptation library and the stereo library are implemented as stand-alone software that *can* be used in conjunction with other pre-loading libraries to, e.g., provide remote visualization with stereo combined (in fact, Figure 3.3 was produced this way); but they equally well can be used for just making applications stereo-aware that do not provide any stereo functionality or to adapt user interfaces of applications actually executed on small-screen devices. Both libraries are, thus, also useful on their own.

Our approach has been picked up by a group of Open Software developers and is now freely available for downloading in an largely extended version [12]. Furthermore, two commercial products—marketed by the companies Sun and Hewlett-Packard [30]—have evolved which greatly resemble the architecture presented in Section 3.2 and 3.3. The latter solution benefits from an optimized, patented image compression algorithm which supports the observation of Section 3.2.3 that image compression is the most effective and most desirable optimization in remote visualization environments.

Chapter 4

Grid Resampling

In the previous chapter techniques were presented that allow for analyzing data sets located on remote hosts. The discussion assumed that the remote host, the render server, possesses sufficient resources—regarding both the CPU and the GPU—such that the data can be visualized at interactive framerates. Furthermore, it assumed that a decent network connection is available for transmitting the final images to the interaction server, i.e. the user. However, there is no necessity that these prerequisites are met. Thus, a researcher may be forced to analyze the data locally using whatever hardware and software is available.

This especially applies to industrial flow visualization. In these environments, data sets often result from numerical simulations. As was argued in Chapter 3, these simulations often result in tremendous amounts of data. Even worse, these data are often defined on unstructured grids to smoothly adapt the simulation volume to potential device geometries. As seen in Section 2.2, unstructured grids are particularly hard to handle due to their irregular topologies. Thus, visualizing unstructured grids puts high demands on both the computing and graphics resources of a computer system—demands that locally available hardware often cannot fulfill, thereby making the analysis of data difficult.

On the other hand, we saw that operations on Cartesian grids are simple and efficient. Accordingly, Cartesian grids can also be visualized efficiently and allow for interactive framerates even for data sets consisting of millions of cells.

In this chapter we present algorithms for converting unstructured grids to a hierarchy of Cartesian grids, work that was driven by actual demands of the car manufacturing industry and conducted in close cooperation with an industrial partner (BMW AG). In contrast to the approaches presented in the previous chapter striving for genericness, the solutions discussed in this chapter are, thus, tailored to a specific scenario. Of course, converting data also results in an inherent loss in accuracy (the approach thus contrasts remote visualization). As will be shown, however, this loss is acceptable. Data conversion thus poses another possibility

for accelerating flow visualization in environments where neither distant resources for remote visualization nor local resources for on-site visualization of the original data are available. The results were first published with M. Schulz [98].

4.1 Flow Simulation Packages

Most car manufacturers use different kinds of numerical flow simulation packages since neither software is most accurate in any part of the vehicle—some result in higher accuracy in the engine compartment, some are better in predicting the flow around the car body.

One package, *StarCD*, starts with the Navier-Stokes equations for describing the flow. By discretizing the differential equations, a representation is obtained that can then be solved iteratively (usually using supercomputers). StarCD solves the Navier-Stokes equations on unstructured grids enclosing the vehicle geometry. A typical unstructured grid as it is used by car manufacturers is shown in Figure 4.4. As a result, a visualization tool capable of visualizing StarCD data must also be able to process data defined on unstructured grids. *ProStar*, the visualization tool provided with StarCD, has this capability. However, since the algorithms required for visualizing unstructured data are inherently complex, ProStar usually does not allow for interactive work for typical data sets of the automotive industry comprising millions of cells.

A second package, *PowerFLOW*, uses a modification of *lattice-gas automata* (LGA) [71]. While flow simulations based on the Navier-Stokes equations ignore the fact that matter has some structure in order to be able to describe the flow by differential equations, LGA simulations do consider this quantification and simulate the fluid flow by computing the particle-particle interactions found in physical gases and fluids. Since these interactions in general are very complex, LGA simulations confine particle positions to a discrete grid (hence the name of the method). These grids, in turn, for simplicity are usually made up of cuboids independent of any vehicle geometry the fluid is flowing around. Accordingly, data resulting from LGA simulations are considerably simpler to visualize. The modified approach used by PowerFLOW, *lattice-Boltzmann simulations*, essentially uses identical grids but works with distribution functions instead of binary particles, yielding less noisy data (Section 5.1).

For both simulation types, higher accuracy can be achieved by reducing the cell size. Unstructured grids allow for a smooth transition from large cell sizes to small cell sizes (Figure 2.3, right). Furthermore, grid operations like computing derivatives remain unaffected by local refinements. For LGA simulations, however, cell size reductions break up the regular topology inherent in regular grids and also require modifications of the grid operations. An example for the

resulting grid hierarchy was already shown in Figure 2.5. Although hierarchical grids complicate matters, they are nevertheless used in current LGA simulations to make the computations more efficient by spending most of the processor time on regions of high interest to the researcher. Accordingly, the corresponding visualization software must be capable of handling grid hierarchies. Details about visualizing hierarchies of Cartesian meshes can be found in [84], work that laid the basis for *PowerVIZ* [85], the visualization system provided with PowerFLOW already seen in previous chapters. PowerVIZ generates pleasing visualizations at interactive framerates even for the largest data sets.

The bottom line is that engineers have to use several solvers and several visualization tools. The former is easily justified by improved accuracy of the flow simulations. For the latter, however, having to train employees for several visualizations tools—one of them not even allowing for interactive work—is highly unfavorable from an economical point of view. Even worse, the training is spent also on tools exhibiting very low performance in numerical flow visualization. Thus, a solution is needed that enables engineers to keep employing several flow solvers but at the same time to concentrate on a single visualization tool capable of visualizing data from different sources. For this work, the part of the visualization software is taken by PowerVIZ.

4.2 Existing Resampling Solutions

Obviously, the scenario developed in the previous section—using PowerVIZ for visualizing data defined on different grid types—either requires extending PowerVIZ about algorithms capable of processing unstructured grids, or it requires converting third-party data to the format understood by PowerVIZ. Considering that the performance of PowerVIZ must be ascribed mainly to the more efficient algorithms on hierarchical grids, the latter alternative is to be preferred. Furthermore, since the software is distributed commercially, source code modifications are difficult, anyway, while, on the other hand, *resampling* the unstructured grid (as conversions between different grid types are commonly referred to) is completely non-intrusive.

Resampling unstructured grids is a technique often employed for rendering unstructured volume data sets. In [1] a grid adaptation module for multi-level Cartesian flow solvers is presented. However, the focus of the paper is not on the visualization part but rather on flow solver characteristics and potential inaccuracies introduced by the resampling step. In contrast, the research presented in [105] and [108] exclusively discusses visualization-related issues. Both papers perform the resampling step with the help of graphics hardware, the former publication splitting the workload between the GPU and the CPU. In either case, refinement

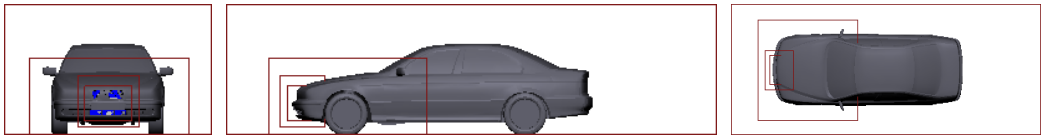

Figure 4.1 Actual, manually defined refinement regions for a BMW 528i vehicle. Data set courtesy BMW AG.

levels must be known a priori and specified by the user. The same is true for the software presented in [45] which provides interactively manipulable cutting planes on which the enclosing unstructured volume is resampled.

4.3 Semi-Automatic Resampling

The PowerVIZ software requires input data to be defined on a hierarchy of Cartesian grids (including a single Cartesian grid). Utilizing a single Cartesian grid is only acceptable if any part of the flow is equally interesting. In general, this cannot be assumed since most flows include both turbulent regions where velocities vary considerably even over small distances, and regions where velocities defined for neighboring grid points are virtually identical over large distances (we then speak of *laminar* flows). Turbulent regions carry the bigger part of the kinetic energy in a flow and, thus, usually are of higher interest to the researcher than laminar flows (in Chapter 5 we will discuss techniques for approximately identifying turbulent regions). However, "interesting" is not necessarily equivalent to "turbulent" as can be observed by considering a CFD simulation of an engine compartment. For example, the degree to which an engine is cooled considerably depends on the amount of air flowing through inlets installed at the vehicle front. Thus, in order to obtain accurate quantitative data about the cooling, the computational grid around the air inlets will have to be finely resolved. However, this very region is not necessarily turbulent which illustrates that grid generation in general is application-dependent and should incorporate domain knowledge. A sensible approach for constructing the required hierarchy of Cartesian grids is, therefore, to let engineers define *refinement regions* (Figure 4.1) and corresponding *refinement levels* manually and to automatically perform the grid conversion based on this information. We will call this approach *semi-automatic resampling*.

4.3.1 Algorithmic Overview

The main task of a resampling algorithm as it was defined above is to interpolate data values. For each interpolation, the algorithm needs to find a pair of cells—

one cell from the source grid (the unstructured grid) and one from the destination grid (the hierarchy of Cartesian grids).

When iterating over the destination cells, the destination cell is known and the source cell has to be determined. Since data values are interpolated at a given point in space, this requires determining the grid cell of the source grid which encloses the center of the given destination grid cell. This task is commonly called *point location* and time-consuming if no neighbor information is available for the unstructured grid [80].

On the contrary, when iterating over the source cells, finding the destination grid cells covering the given source cell is trivial if the destination grid is Cartesian since the point location reduces to a simple truncated division. This benefit is independent of any neighbor knowledge; therefore, iterating over the source cells—a so-called *object-based approach*—is to be preferred and, accordingly, used in this work.

In any case the resampling algorithm requires knowledge of both the unstructured grid data (cell definitions and data values) and the geometry of the resampled grid. As mentioned, for the semi-automatic approach the resampled grid is obtained by manually defining pairs of cuboids and refinement levels. To access the unstructured grid data, we take advantage of a ProStar feature allowing for an export of grid data in CGNS format [67], an open standard for the exchange of computational fluid dynamics data that comes with a variety of libraries for reading, writing, and manipulating CGNS files.

Using the object-based approach described above, the actual resampling is straightforward (Figure 4.2). The outermost loop iterates over the cells of the unstructured grid. Each cell is first subdivided into tetrahedra to unify cell treatment in the following steps. Since a tetrahedron may intersect several refinement regions, calculating these intersections is the next step. The algorithm then calculates a bounding box for each intersection volume and tests each grid cell of the bounding box against the tetrahedron. This inside test is implemented using barycentric coordinates. If the solution data, i.e. the data attached to a grid point or grid cell, are vertex-based, the same barycentric coordinates can then be used to linearly interpolate data values at the cell center from the data values defined at the surrounding tetrahedron vertices. If the solutions are cell-centered, nearest neighbor interpolation is used (see Section 2.2.2).

4.3.2 Evaluation

The semi-automatic resampling algorithm was evaluated using real-world data sets of the car manufacturing industry with 12-element solution arrays defined for each cell. The refinement regions were defined manually by practitioners. The results show that, on the one hand, accuracies with average relative errors

```
U ← cells of unstructured grid
C ← cells of Cartesian grid
R ← refinement regions

for each cell ∈ U do
    T ← {tetrahedra of cell}
    for each tetrahedron ∈ T do
        I ← {r ∈ R | r intersects tetrahedron}
        for each region ∈ I do
            C ← {c ∈ C | c inside intersection}
            for each cell ∈ C do
                B ← {bary. coordinates of cell center}
                if ∀b ∈ B : 0 ≤ b ≤ 1 do
                    interpolate linearly with weights B
                    save interpolated data to disk
                end if
            end for
        end for
    end for
end for
```

Figure 4.2 Algorithm for semi-automatic resampling of unstructured grids using manually-defined refinement regions.

of less than two percent for velocity magnitude can be obtained with appropriately defined refinement regions. For the data set shown in Figure 4.4, left, which comprises 9,360,788 cells (car body plus engine compartment), however, 80 percent of the cells comprising the original grid are missed despite a resulting cell count of 7,205,311; thus, the resampled data set includes only about 20 percent of the original information although a cell reduction to only about 77 percent is observed.

A qualitative comparison of isosurfaces extracted from the original grid using ProStar and the resampled grid using PowerVIZ is shown in Figure 4.3 by example of a 448,450 cell data set whose resampled version comprises 321,451 cells. Using ProStar for visualizing the unstructured data set, rotating the isosurface or moving a slice are highly non-interactive (0.9 fps and 0.3 fps, respectively[1]). While the former is a mere implementation issue, the latter is a direct consequence of the complex grid topology. Using the resampled data set and PowerVIZ, the framerates increase by several factors: Rotating the isosurface is now accomplished at

[1] AMD Athlon 1.2 GHz, 512 MB main memory, NVIDIA GeForce 3, 1600×1200 viewport.

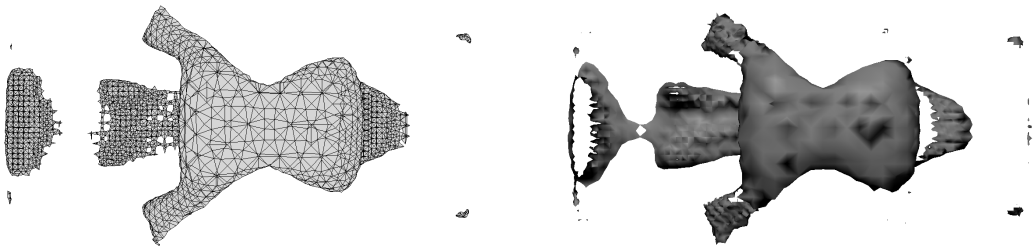

Figure 4.3 Isosurfaces of velocity magnitude of a flow around a car body. Left: original unstructured grid (ProStar); right: semi-automatically resampled grid visualized with PowerVIZ. Vehicle geometry data are not available for ProStar and, thus, are not shown for PowerVIZ either.

61 fps, moving a slice at 1.8 fps. Although the latter fails to allow for interactive work, it nevertheless results in a significant performance gain. Regarding the acceleration of numerical flow visualization, semi-automatic resampling thus serves its purpose. The observed loss of information which is also indicated by significant differences in the isosurfaces shown in Figure 4.3, however, suggests that manually-defined refinement regions may be extremely inefficient with regard to the number of grid cells and the resulting accuracy and that the semi-automatic approach has to be reconsidered.

4.4 Adaptive Resampling

The results of Section 4.3.2 show that manually-defined refinement regions bare the risk of missing details of the unstructured grid. Of course, the implementation can inform the engineer about the number of cells not considered for the resampling process, allowing the engineer to adjust the bounding boxes or refinement levels accordingly until the desired result is obtained. However, this iteration may become tedious if the resampling is time-consuming and it can easily lead to an unnecessarily large number of grid cells. Thus, the question poses as to how a suitable hierarchy of Cartesian grids can be constructed automatically.

One common approach also used in [1] is to increase the number of cells until a property of the unstructured grid (gradients of pressure, velocity magnitude, etc.) falls below a given threshold. These algorithms are independent of the geometry of the unstructured grid and will always produce trustworthy results. However, if several scalar properties are defined on the data set, calculating gradients becomes very expensive. In addition, these so-called *error-based* algorithms require several vector field properties to be available in memory while constructing the hierarchy since in general it cannot be assumed that the optimal hierarchy for one prop-

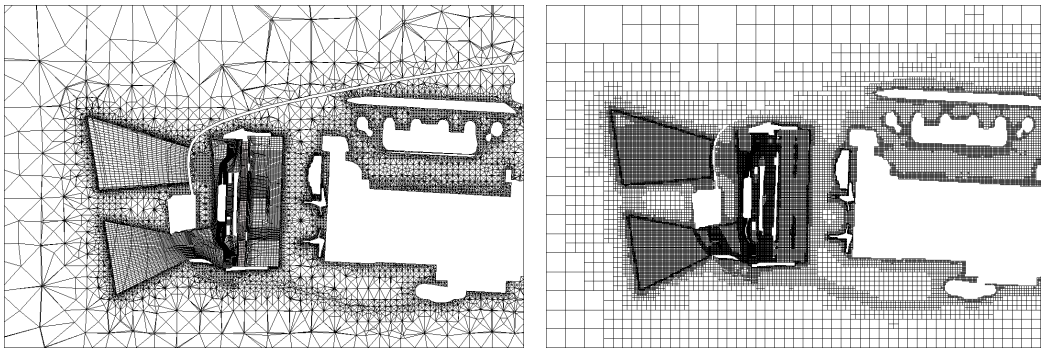

Figure 4.4 Original unstructured grid and corrected octree structure. The images show the engine compartment of a BMW 528i vehicle. Data set courtesy BMW AG.

erty will necessarily be identical to those constructed for other properties. Thus, error-based algorithms put high demands on both the host performing the actual resampling and the client used for visualizing the data since possibly several grids are generated.

For this work, we therefore propose a *geometry-based* approach. The approach bases on the assumption that an unstructured grid to be resampled has been prepared with great care and that, therefore, regions of small cell sizes reflect regions of high gradient magnitude. Moreover, as was argued in Section 4.3 by the example of engine cooling, any region of high interest to the engineer will generally be resolved finely. Thus, results comparable to those of error-based algorithms can be obtained at reduced costs with regard to both CPU time (since no error metrics must be computed) and memory consumption (since no solution data must be kept in memory for constructing the grid hierarchy).

The basic idea of the adaptive resampling algorithm devised for this work can be described by a three-phase process: first, an octree [82] is constructed whose leaves are approximately the same size as the unstructured grid cells covering the respective leaf; second, the octree structure is corrected; third; the solution data is interpolated.

Intuitively, this algorithm does *not* result in refinement regions as defined above. However, an octree is just a special, if also very inefficient, case of refinement regions all comprising only a single grid cell. Thus, PowerVIZ must be capable of processing a regular octree structure, too. And, in fact, PowerVIZ has this capability. Yet, to speed up computations, a volume growing algorithm is first applied to cluster regions of similar cell sizes. In practice, this approach works well and relieves us from the burden of having to construct refinement regions (in the intuitive sense) ourselves. The downside, however, is that PowerVIZ is only capable of visualizing octree structures artifact-free when being *restricted*,

i.e. when levels of neighboring leaves do not differ by more than one level. This explains step two—the correction step—of the three-phase resampling process.

4.4.1 Octree Construction

To construct the octree, a root region is created first. The region is accompanied by a list of indices of those cells intersecting the region. Initially, the list contains the indices of all cells. In the next step, an average edge length of the intersecting cells is calculated. Moreover, to reduce the influence of degenerated cells, i.e. cells with highly varying edge lengths, the average length of each cell is multiplied by the ratio of the shortest to the longest cell edge. If the weighted average edge length is smaller than the cell size, the region is subdivided into octants and new index lists are compiled. The processing then continues recursively. When the recursion terminates, a grid cell—or rather: a one-cell refinement region—of the size and position of the octant is generated. A comparison between the original grid and the grid resulting from this refinement criterion is shown in Figure 4.4.

Mapping octree leaves to grid cells makes an octree a very natural choice for the grid generation. However, if the resampling is supposed to be operable also on low-resource platforms, it is required to have moderate memory consumption. Unfortunately, conventional pointer-based octrees are not memory efficient since memory is allocated both for the leaves (grid cells) as well as for intermediate nodes. To overcome this problem, a linear, pointer-less octree approach was chosen based on the work presented in [24] which discusses a set of algorithms that are efficient but nevertheless easy to implement. However, the presented algorithms are optimized for speed rather than for memory efficiency and again use pointers to store the tree hierarchy. The implementation used in this work is still efficient but does not require pointers and it can be easily saved to disk; thus, it allows the user to confine the memory requirements during the octree construction. The fundamental concept of the approach—the concept of *locational codes*—and the accompanying notation, however, remain untouched.

Definitions and Notations

For an octree of n levels, a locational code is defined as a three-element vector of n-bit strings that represent the branching pattern followed while recursively refining the root region. The first bit of each element is always 0 and denotes the root of the octree. The remaining bits are filled from the left (most-significant bit) to the right (least-significant bit). Thus, under the assumption that levels are counted bottom to top starting with 0, bit k is set at level k. Figure 4.5 shows a level 5 quadtree—the 2D equivalent of an octree—and the corresponding locational codes in each of the dimensions X and Y for a selected leaf.

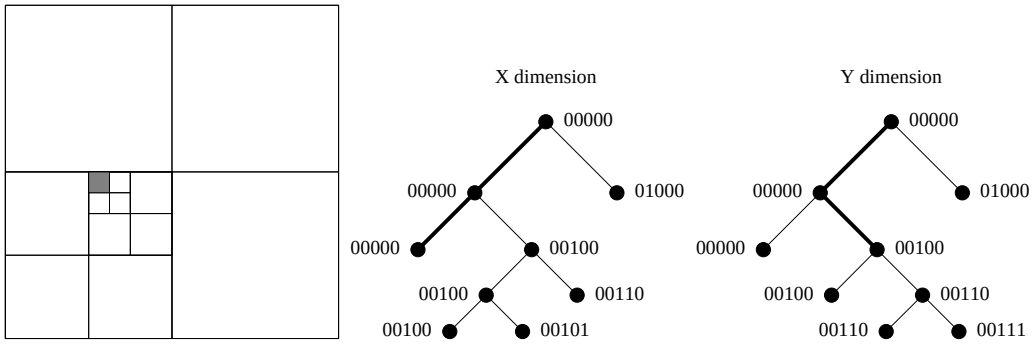

Figure 4.5 Quadtree of level 5. The locational code of the marked quadrant is (00100, 00111). The locational codes of the quadtree are shown on the right.

Since CPUs can only handle data types with an integral number of bytes, some memory is wasted if the octree has only five levels as in the example. For the example tree, at least one byte has to be reserved for each element of the locational codes. In practice, 32 levels are sufficient for all the data sets in the automotive industry; thus, locational codes can be expected never to require more than $3 \times 4 = 12$ bytes.

Linear Octree Construction

Our modified approach requires the locational codes of all grid cells (leaves in the octree) to be saved in an array $\mathcal{L}_\mathrm{C}$. Each entry in the array must also contain a pointer to the application data linked to the cell. In addition, for each dimension another vector $\mathcal{L}_\mathrm{D}$, $\mathrm{D} \in \{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$, is required that stores the distinct elements of the locational codes in this dimension and the minimum (farthest from the root) levels. All arrays must be sorted according to increasing locational codes (any dimension in case of the array $\mathcal{L}_\mathrm{C}$ that holds the complete codes). For the example of Figure 4.5, left, the arrays shown in Table 4.1 are obtained.

Finding Neighbors

Finding neighbors of a given grid cell is the only octree operation required for this work. The algorithm described in [24] first calculates a *guess* for the locational code of the relevant neighbor. For neighbors to the right, top, or front of the cell (i.e. neighbors with increasing X, Y, or Z coordinates) this guess is obtained by adding the binary representation of the grid cell size (2^{level}) to the corresponding element of the locational code. That is, in order to find the right neighbor of the grey square in Figure 4.5, left, the guess is obtained by adding (00100, 00111) and (00001, 00000), yielding the locational code of the right neighbor. However,

Table 4.1 Supporting data structures $\mathcal{L}_C$, $\mathcal{L}_X$, and $\mathcal{L}_Y$ for the locational codes of the octree shown in Figure 4.5.

$\mathcal{L}_C$	
X	Y
00000	00000
00000	00100
00000	01000
00100	00000
00100	00100
00100	00110
00100	00111
00101	00110
00101	00111
00110	00100
00110	00110
01000	00000
01000	01000

$\mathcal{L}_X$	
X	Level
00000	2
00100	0
00101	0
00110	1
01000	3

$\mathcal{L}_Y$	
Y	Level
00000	2
00100	1
00110	0
00111	0
01000	3

finding the locational code of a neighbor is not always as easy: Adding (00100, 00111) and (00000, 00001) yields (00100, 01000), which is a locational code that does not match any grid cell. Finding the top neighbor, therefore, requires further traversal of the octree.

For the left and bottom neighbors, the problem is even harder. While we can be sure to hit a cell boundary by *adding* the grid cell size to the locational code, there is a high probability that we will not hit a boundary when *subtracting* the cell size. However, the distance to the next boundary will be at least the size of the smallest grid cell, namely $2^0 = 1$. Thus, the neighbor can be found by subtracting 00001 from the locational code and—using this locational code as a template—traversing the octree down from the root until a leaf is reached. Accordingly, to find the bottom left neighbor of the grey square, (00001, 00001) is first subtracted from (00100, 00111) to obtain the template (00011, 00110). Neither of the two elements denotes the boundary of any cell; thus, the octree is traversed downwards until level 2 is reached. The path is indicated by thick lines in Figure 4.5, right.

Using the adapted data structures, the traversal can be simulated using solely the locational codes of the octree leaves. The first step in the algorithm for obtaining an initial guess remains the same as before. In the second step, the lowest n bits of the guess are cleared to obtain templates, starting with the level of the root node and decreasing n repeatedly until the level is less than the maximum

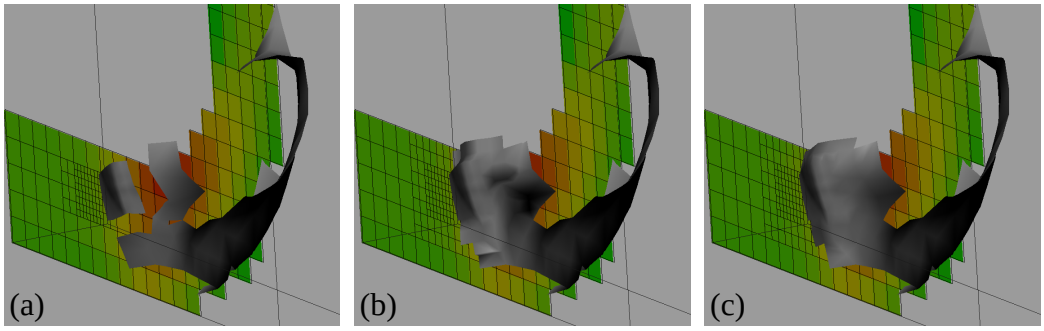

Figure 4.6 Comparison of isosurfaces calculated by PowerVIZ. **(a)** Uncorrected octree, **(b)** corrected octree without interpolation (subdivision only), **(c)** full correction with interpolation.

of lowest levels of the three templates. This information can be obtained easily since the lowest levels are part of the data structure and binary searches in $\mathcal{L}_D$ only consider distinct locational codes. Since each triple of templates denotes a potential neighbor, the triple is then used for a binary search in $\mathcal{L}_C$ to determine whether the neighbor exists.

Considering again the problem of finding the top neighbor of the grey square in Figure 4.5, we thus restart with the guess (00100, 01000). The root node has level 5, thus, all bits are cleared to obtain the template (00000, 00000). As is easily seen from the tables $\mathcal{L}_X$ and $\mathcal{L}_Y$, the minimum level for both the X-direction template 00000 and the Y-direction template 00000 is 2. That is, since we are currently examining level 5, we can proceed by checking to see whether a grid cell with locational code (00000, 00000) exists. Since table $\mathcal{L}_C$ approves our guess we assume this cell to be the sought-after neighbor. Proceeding with level 4 we now clear the lowest 4 bits. The resulting template is identical to the previous template and, thus, will not reveal any novel information. For level 3 the template is (00000, 01000). From $\mathcal{L}_C$ we see that this also denotes a valid cell; thus, we replace our neighbor guess by the new template. Furthermore, the maximum of the lowest levels of 00000 and 01000 is 3, the search terminates.

4.4.2 Octree Correction

The construction of octrees using the algorithms described in the previous sections can result in arbitrary level transitions, i.e. cells may be adjacent to cells four times the edge length of their own or above. These transitions do not affect the interpolation step but they may introduce artifacts in PowerVIZ if not handled correctly (Figure 4.6 (a)). Of course, special handling of so-called *unrestricted octrees* adds additional complexity to the visualization algorithms and results in

increased execution times [103]. Therefore, removing transitions of more than one level only once during the resampling instead of each time the data set is visualized is more appropriate. And as can be seen in Figure 4.6 (b), the correction not only requires adding grid cells but also interpolating new data values to obtain a well-defined isosurface.

A naive implementation for removing invalid level transitions in an octree iterates over the octree leaves. For each leaf, the neighbor leaves are determined and the level information compared. If the level difference is greater than one, the leaf closer to the root node is subdivided into eight subleaves. The algorithm then proceeds with the next leaf and terminates if a leaf is visited again and no level transitions had to be removed since the last visit. This algorithm is very robust but suffers from very high execution times.

The observation that small grid cell sizes are propagated upwards in the octree leads to a modified algorithm. Assuming all cells of the lowest level (0) are processed first, extra cells are added if and only if a leaf of level 2 or a higher level is a neighbor of a leaf being processed. As a result, all cells added to the octree are at least level 1. When continuing processing grid cells of the next level (1), there is no possibility that again grid cells of level 1 are added to the tree because level 1 can only be obtained if a cell is adjacent to a cell of level 0—and these were just processed. Thus, after the first iteration all grid cells of level 1 and below are known, and after the second iteration all cells of level 2. In general, after the nth iteration all cells of level $\leq n$ are known. This suggests that the leaves should be bucketed first by their levels and then processed from lower to higher levels. If leaves need to be subdivided, the subleaves are added to the appropriate bucket. Using this approach, no leaf has to be processed twice. However, the octree information generated in phase one is stored in files and cannot be accessed easily for removing subdivided leaves. In addition, if high level transitions have to be removed, significantly more memory is required.

To cope with these problems, a simplified octree correction is used in this work demanding that invalid level transitions can only be resolved by completely refining the larger grid cell. As illustrated in Figure 4.7, this approach inherently results in an increased number of additional cells. On the other hand, complete refinement allows for a more efficient encoding of additional voxels since only the number of additional levels needs to be stored for each subdivided grid cell. This number, in turn, will not become large due to exponential growth in the number of cells accompanying the refinement; thus, a single *refinement byte* easily suffices for storing the additional refinement levels.

Regarding the actual refinement step, if the refinement byte of a grid cell is *not* set, the usual 26 neighbors (assuming non-border cells) have to be located. If, on the other hand, the refinement byte is set, the cell must be handled like a cluster of cells of a lower level—obviously, only subcells of these clusters defining the

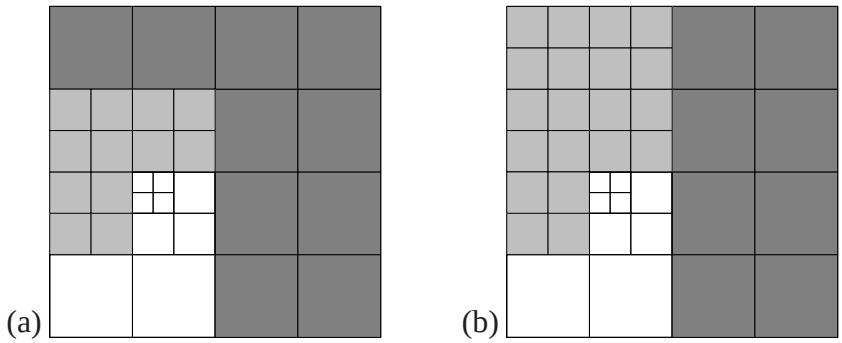

Figure 4.7 Comparison of correction methods at the example of the quadtree of Figure 4.5, left. **(a)** Optimal correction strategy, **(b)** simplified correction strategy.

cluster surface may call for subsequent refinements. If an invalid transition is detected, the refinement byte of the neighbor cell is appropriately adjusted and the refined cell added to the corresponding bucket to ensure that the level propagation can continue correctly. Figure 4.6 (c) shows the final result of the correction.

4.4.3 Interpolation

The data interpolation phase is the last phase of the resampling process. Since all the information necessary for the interpolation is being collected in the preceding stages, the interpolation phase is straightforward.

In the octree construction phase, the grid information is written to disk in several files (the total number of grid cells is unknown in advance). For each grid cell, the locational codes, its level, the index of the surrounding cell of the unstructured grid, the number of intersecting cells, and the indices of the intersecting cells are saved. The files are processed independently one at a time. For each grid cell stored in the files, the number of additional refinement levels, i.e. the refinement byte, is looked up and—if set—the cell is subdivided. Using the solution data of the unstructured grid, the scalar properties of the cell can then be interpolated.

For vertex-based solutions, the value is interpolated using the information of the surrounding element. For cell-centered data, we take advantage of information about intersecting elements to interpolate a data value by inverse distance weighting (Section 2.2.2).

4.4.4 Parallelization

Compared to semi-automatic resampling, the adaptive resampling approach is considerably more complex. Thus, while the former way is passable even for large data sets of millions of cells, the latter—especially the octree correction as

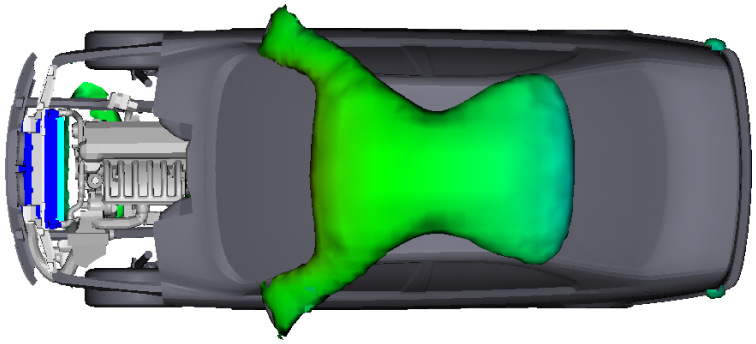

Figure 4.8 Isosurface computed with PowerVIZ using the adaptively resampled grid. The scalar property is velocity magnitude, the isovalue matches the one of Figure 4.3.

an analysis shows—quickly becomes prohibitively expensive. The adaptive resampling algorithm, therefore, has also been implemented to take advantage of multi-processor machines. Each phase was adapted separately and independently of any other phases. Like the parallel program introduced in Section 3.3.3, the implementation is based on POSIX pthreads.

In the octree construction phase a pool of worker threads is created first. When an octant is required to be resampled, it is tried to find an available worker thread. If a worker thread can be found, the responsibility for refining the octant is transferred to the thread which then operates on its own. As soon as the assigned octant is resampled (this includes the assignment of suboctants to other threads), the thread puts itself back into the worker pool. If no thread can be found, non-blocking wait must occur. The reason for this is that otherwise the number of threads required to prevent deadlock situations will grow exponentially (at the nth level, 8^n new threads have to be available); thus, letting a thread do the work on its own is preferable over a blocking wait. Per-thread working areas are used to minimize the number of synchronization operations while generating grid information. A single lock is used for synchronization.

In the octree correction phase, again the thread pool concept is employed. In this phase, work packages of grid cells to be tested for invalid transitions are assigned to worker threads. In contrast to the preceding phase, a blocking wait is used for thread selection to avoid situations where the assigning thread is working on its own and, therefore, cannot assign new work packages to idle threads. Since the processing of a bucket has to be finished before continuing with the next one, joining the threads after the last work package has been assigned is indispensable, albeit this inevitably leads to a stepwise reduction in the degree of parallelization towards the end of the processing of a bucket. However, by keeping the work packages small (in the order of several thousands of grid cells per thread), this

phase can be kept rather short.

Obviously, when a refinement byte needs to be adjusted, a race condition may occur. Therefore, a critical region must be established. Using one semaphore or mutex per cell certainly allows for a high degree of parallelization but it also requires a lot of memory (the respective structure `sem_t` occupies 16 byte on PCs and Linux and 64 byte on an SGI Onyx and IRIX). Using only one lock for all the grid cells requires a minimum of memory but shows a severe competition between the threads and a CPU utilization of no more than 200 percent even on four-processor machines in experiments. A good trade-off is to create a small number of locks (e.g. 256) and then to use the original grid cell index and a modulo operation to assign locks to cells. With this solution, conflicts are reduced dramatically and full parallelization of the correction stage becomes possible. For updating buckets, a similar approach using the level of the grid cell of a bucket is used.

In the last stage, thread assignment takes place on the basis of complete files written during octree construction. Again, competition for locks has to be avoided when accessing the global data structures (number of grid cells, pointers to interpolated cell data). As in the octree construction phase, per-thread working areas are allocated which are regularly synchronized with the global structures.

4.4.5 Results

In its original (sequential) implementation the adaptive resampling algorithm is about an order of magnitude slower than the semi-automatic approach (134 s vs. 12 s for the car body data set of 448,450 cells, hardware configuration as used in Section 4.3.2). The ratio of the number of grid cells comprising the original grid and the number of cells comprising the resampled grid in general is unpredictable and varies by more than 50 percent in both directions. Nevertheless, a significant accuracy increase can be observed yielding average relative errors of less than one percent. Maximum relative errors, on the other hand, still range in the order of 50 percent at worst.

Figure 4.8 shows another screenshot of the isosurfaces already seen in Figure 4.3. Clearly, the isosurface part on top of the vehicle is significantly smoother (and, hence, closer to the original) despite a further decrease in the number of grid cells to 245,585 cells—a strong argument in favor of fully-automatic resampling. The reduced number of cells also proves advantageous for the visualization: rotating the isosurface now yields a slightly increased framerate of 72 fps.

Regarding memory consumption we observed sublinear growth with the number of unstructured grid cells. The maximum physical memory consumption for resampling the 30 MB car body data set during the octree construction is about 110 MB, during the octree correction 68 MB, and during the interpolation phase

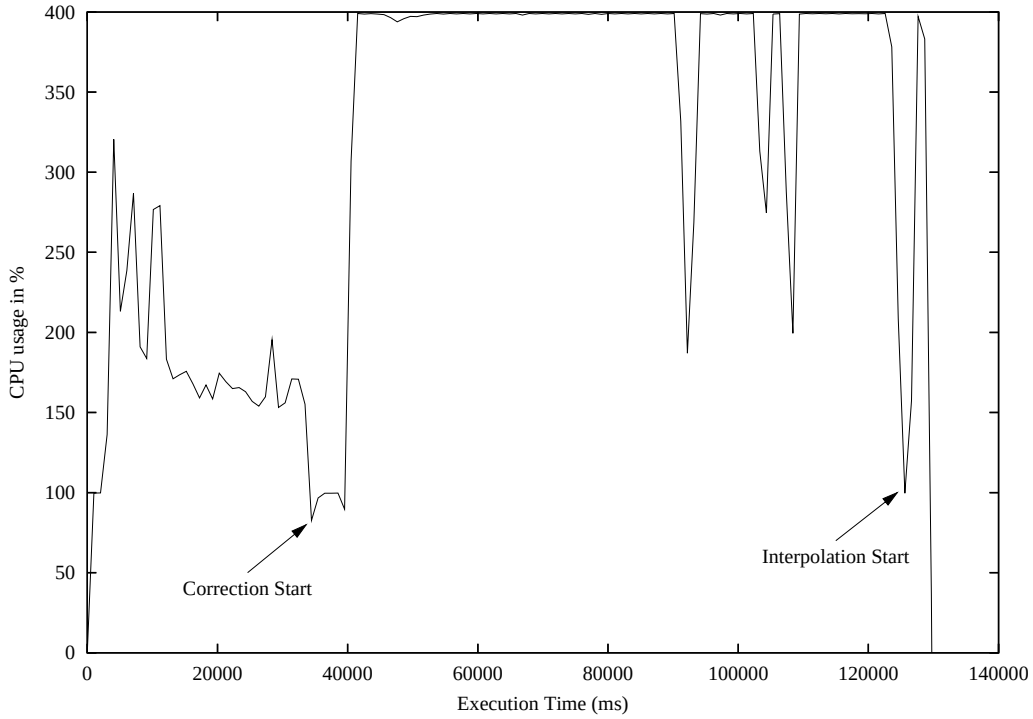

Figure 4.9 CPU-usage of the parallel, fully-automatic resampling algorithm measured on a four-processor SGI Onyx.

128 MB. For the 759 MB car data set including the engine compartment, the values increase to 866 MB, 1,080 MB, and 1,622 MB, respectively, on an SGI Onyx. Due to an extensive use of memory-mapped files, the actual physical memory requirements can be expected to be much lower than virtual memory consumption.

Using the parallelized version, speed-up factors of about three can be measured on a four-processor SGI Onyx. The reason for not reaching full parallelization is revealed in Figure 4.9: Although private working areas are available per thread, the refinement phase does not parallelize well. In fact, even for data sets of high octree depths CPU utilization of nearly 400 percent is rarely reached. However, for large data sets, the octree correction becomes the dominating part and speed-ups by almost a factor of four can be observed, thereby making the time invested into the data resampling bearable.

Considering further the long simulation times involved in the flow data generation and the fact that resampling is typically performed only once, adaptive resampling presents a valuable, albeit highly non-interactive, tool for performing quantitative analyses of large, unstructured data sets. Whenever pre-processing times are non-critical, it should be favored over less effective and accurate semi-automatic resampling.

Chapter 5

Feature Extraction

The previous chapters introduced acceleration of numerical flow visualization by means of remote visualization and grid resampling. While the former may not be usable in certain environments due to missing resources (like high-speed network connections), the latter may not be desired due to an inherent loss in accuracy. Furthermore, since resampling still presents the original data, the extent to which the visualization can be accelerated by grid conversions is limited, anyway.

Involved in this issue is also the fact that both remote visualization and grid resampling are mere visualization-related acceleration techniques. More favorable are methods that accelerate both the visualization and the data analysis since this is the ultimate goal of numerical flow visualization. Flow field features have the potential of accomplishing this task.

As was stated, the only flow field feature relevant for this work are vortices, i.e. regions of spiraling flow. The method currently considered most effective in accurately identifying these vortical regions in incompressible flows is the λ_2 method introduced in Section 2.3.2. Nevertheless, the method is not flawless: shortcomings have hitherto been observed in turbomachinery flows (i.e. flows in turbines and engines) [78] and flows with strong axial stretching [110]; and some more critical work [68] even claims that *"the only consequence that can be drawn from the definition of Jeong and Hussain (1995) is that, if in a region we have* $\lambda_2 < 0$*, then for all the points of that region the vorticity predominates over the strain in a 2D infinitesimal neighborhood of the point"* and closes with the proposition that *"in general the definition ... does not say anything about the existence, or not, of a vortex."* Work of collaborating fluid dynamics engineers [52] showing great similarity between physically visualized Λ-vortices and Λ-vortices extracted numerically from simulation data indicates that the λ_2 method is very capable of detecting vortices in certain flow fields indeed. In fact, not accuracy or reliability seem to be issues but rather two other significant drawbacks: First, the method is computationally expensive due to complex eigenvalue calculations and, second,

while the method informs us about any vortices contained in the flow, it fails to extract a separated list of vortices. For analyzing flows the former merely makes the analysis more time-consuming; the latter, however, makes certain analysis tasks of fluid dynamics engineers like the study of vortex interactions in vortex dynamics completely impossible.

It goes without saying that working with computer software is the more tedious the more computationally expensive the required calculations are; thus, it is obvious that high execution times pose a drawback of the λ_2 method. In general, however, one would expect that for a given vector field there is no need to extract vortices several times. Vortex extraction, therefore, will generally be considered a one-time pre-processing step—similar to grid resampling—whose computational complexity is of minor significance. The whole extent of high execution times, however, is only revealed when reconsidering flow simulation techniques (Section 4.1 already gave an introduction) used in practice and the properties of these techniques and the resulting data. In this chapter, we will therefore first revisit flow data acquisition methods in fluid dynamics research before elaborating on acceleration and segmentation techniques and techniques for improving the visualization and interaction with the extracted structures. For all implementations, the velocity field is assumed to be defined on a regular grid possibly generated using the techniques described in Chapter 4 by resampling the original grid to a hierarchy of Cartesian grids comprising cells of only a single level.

5.1 Flow Simulation Revisited

Due to the complexity of the eigenvalue calculations involved in the computation of the λ_2 scalar field, the quality of the vortex extraction strongly depends on the quality of the input vector field. The term “quality” in this context refers to the presence or absence of noise, i.e. high-frequency fluctuations in flow quantities. To obtain optimal results, this noise needs to be removed from the input data by filtering before applying the λ_2 method since otherwise the vortex extraction will be unreliable and produce a puzzling lot of small-scale vortex structures making it hard for the engineer to spot the relevant structures (Figure 5.1). Ideally, if the frequency of the noise is known in advance, a bandpass filter is designed capable of removing any interfering frequencies. If, however, the noise cannot be exactly located in the frequency domain, obtaining the optimal visualizations requires several iterations of a cycle of denoising, vortex detection, and evaluation. And even if the λ_2 computation only takes some seconds, analyzing the data becomes a very tedious task. Thus, fast λ_2 computations are highly welcome.

Obviously, as any data acquisition technique involving physical measurements is subject to noise, the above arguments apply to all data sets obtained by physical

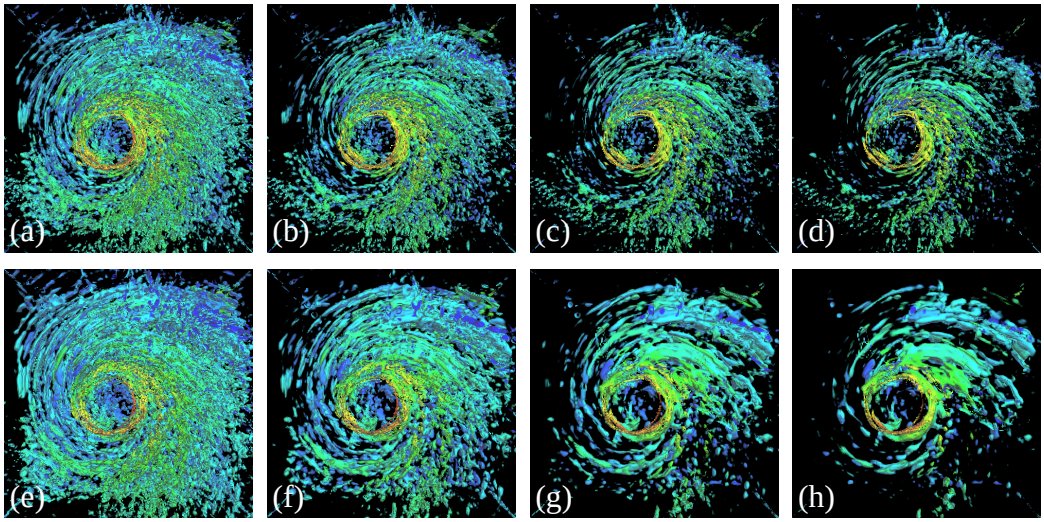

Figure 5.1 λ_2 isosurfaces of the hurricane Isabel data set. Images **(a)–(d)** show the effect of successively decreasing the isovalue, images **(e)–(h)** illustrate the effect of applying various filters starting with the same base image. While both adjustments tend to remove small-scale vortex structures, it can be seen that only filtering is capable of extracting relevant, large-scale vortex structures.

flow visualization like LDA or PIV (Section 2.3.1). However, flow data resulting from numerical simulations may be subject to noise, too, as it becomes apparent by examining numerical approaches for computing flows [71, 83].

Many flows can be modeled by the so-called *Navier-Stokes equations*. In theory, flow fields can be computed by directly solving these equations. This approach is known as *Direct Numerical Simulation* (DNS) and it is capable of capturing all scales of motion. Since, however, in turbulent flows there is usually a very wide range of different sizes of motion this means that we require very fine grid resolutions. The required number of grid points is proportional to $\mathrm{Re}^{2.25}$ [83] where Re denotes the *Reynolds Number*, a dimensionless number expressing the ratio of inertial forces to viscous forces and commonly used as indicator for whether a flow is laminar of turbulent. Reynolds numbers span a wide range of values, starting with very low numbers for laminar flows (for the flow shown in Figure 2.6 $\mathrm{Re} = 100$) but easily reaching values of 10^5 for low-viscosity flows; accordingly, in practice, providing for sufficient grid resolution is often too computationally expensive and, thus, frequently disregarded. This, however, eventually leads to underresolved calculations and to different kinds of noise like oscillations when taking derivatives—which is highly troublesome for computing the velocity gradient tensor and, consequently, λ_2 values.

Data obtained by lattice-gas automata (Section 4.1) are subject to (statistical)

noise, too. This is caused by the discrete nature of LGA which requires averaging several grid point values for determining flow quantities. On the other hand, for lattice-Boltzmann simulations this averaging becomes obsolete. The resulting data sets are thus of higher quality than those obtained by LGA simulations.

Also insensitive to noise are *Reynolds-Averaged Navier-Stokes* (RANS) simulations and *Large Eddy Simulations* (LES), both approaches striving to alleviate the high memory and computing costs of DNS by the introduction of *turbulence models*. For the former, all turbulent effects of the flow are modeled and only average flow quantities are captured, i.e. the smallest scales of motion are *never* captured, not even for very high grid resolutions. Since small-scale motion translates to high-frequency changes of flow quantities which, as stated above, characterize noise, data sets obtained by RANS simulations are in general free of noise. Similarly, LES—a hybrid approach between DNS and RANS simulations—extracts the largest scales of motion from the flow field by filtering and models only small-scale structures for which the resolution of the computational grid is insufficient. In contrast to RANS simulations, however, there now is a dependency between grid resolution and the capability of the simulation to resolve small-scale motion. As a result, we can control the amount damping but since high-frequency fluctuations are usually eliminated, the resulting flow fields again are not prone to noise.

The discussion shows that while not all data resulting from flow solvers necessitate filtering, denoising *may* be required for data acquired by *both* experiment and numerical simulation.

5.2 Software Implementation

In Section 2.6.1 it was argued that GPU implementations are highly non-portable and, hence, should be avoided if straight software solutions suffice for attaining the defined goal. For this work, an optimized software was, therefore, implemented for computing λ_2 values based on the Intel Pentium III SSE extension [33]. This extension is capable of processing four-component vectors with 32 bits per component and, therefore, allows for an efficient computation of the velocity gradient tensor $\underline{V}$. In order to make this computation simpler also from an implementation point of view, the software solution proceeds by first adding a one-cell border to the volume defining the velocity field. The solution data of these border cells are chosen such that, when iterating over the original cells, central differences (Section 2.2.1) can be utilized for computing derivatives regardless of whether the respective cell is an interior cell or a cell on the volume surface. Since in the latter case, the derivative must be approximated by one-sided differences, this means that border cell values must be assigned values for which, when

computing derivatives for the neighboring interior cell, the result obtained by central differences equals that resulting from one-sided differences. For any of the original (non-border) cells, the velocity gradient tensor is then computed and the characteristic equation set up and solved, finally yielding the desired eigenvalue λ_2. We will now elaborate on these steps and reuse the results in the Section 5.3.

5.2.1 Computing the Velocity Gradient Tensor

Reconsidering the definition of the velocity gradient tensor $\underline{V}$ given in Equation 2.10, Section 2.3.2, it can be seen that the row vectors comprising the matrix are the gradients of the individual velocity components. Formally: $\underline{V} = (\nabla u_1, \nabla u_2, \nabla u_3)^T$. Accordingly, the matrix columns are vectors comprising the partial derivatives with respect to a single dimension and these, in turn, can be efficiently computed with a single subtraction and a subsequent division (or, equivalently, a multiplication with the reciprocal) using vector operations as supported by the SSE extension. For the first column this translates to:

```
SUB(volume[t][s][r + 1], volume[t][s][r - 1], temp);
MUL(temp, doubleDistsInv, partialsX);
```

where each operand is a four-component floating point vector and `SUB` and `MUL` denote macro definitions for using SSE vector operations. For `MUL`, the macro definition is given by:

```
#define MUL(a,b,c) \
    asm("movups (%0), %%xmm0" : : "r" (a));  \
    asm("movups (%0), %%xmm1" : : "r" (b));  \
    asm("mulps  %xmm0, %xmm1");              \
    asm("movups %%xmm1,(%0)" : : "r" (c));
```

The `ADD` macro is defined analogously. SSE operations usually are advantageous only when used in larger blocks of multiple assembly instructions. However, the computation of $\underline{V}$ simply maps to three pairs of the above `SUB`/`MUL` sequence and, thus, obviously meets this precondition.

5.2.2 Setting up the Characteristic Polynomial

Once the velocity gradient tensor is available, the matrices S and Ω can be computed. These computations are most compactly carried out with matrix processing functions. Expanding the matrix sum $S^2 + \Omega^2$, however, yields:

$$S^2 + \Omega^2 = \left(\frac{\underline{V} + \underline{V}^T}{2}\right)^2 + \left(\frac{\underline{V} - \underline{V}^T}{2}\right)^2$$

$$= \left(\frac{1}{2} \sum_{k=1}^{3} v_{ik} v_{kj} + v_{ki} v_{jk} \right) ,$$

where $\underline{V} = (v_{ij})$. What follows from this is that the matrix $S^2 + \Omega^2$ is symmetric because transposing the matrix simply interchanges the terms which, of course, does not affect the value of the sum. And this, in turn, means that when using matrix manipulation functions some redundant computations are performed. In our implementation, we therefore only compute the six non-redundant matrix entries of $S^2 + \Omega^2$:

```
so00 =  partialsX[0]  *  partialsX[0] +
        partialsY[0]  *  partialsX[1] +
        partialsZ[0]  *  partialsX[2];

so01 =  partialsX[0]  *
       (partialsY[0]  +  partialsX[1])/2.0 +
       (partialsY[0]  *  partialsY[1] +
        partialsZ[0]  *  partialsY[2] +
        partialsX[1]  *  partialsY[1] +
        partialsZ[1]  *  partialsX[2])/2.0;
```

The remaining four matrix entries can be derived analogously by permutations of indices.

5.2.3 Eigenvalue Computation

Given a matrix M and a vector $\mathbf{x}$ having as many rows as M has columns, then λ is called the *eigenvalue* corresponding to the eigenvector $\mathbf{x}$ if and only if $M\mathbf{x} = \lambda\mathbf{x}$. If we move the right hand side of the equation to the left, we obtain a linear system of equations $(M - \lambda E)\,\mathbf{x} = 0$ where E denotes the unity matrix. This linear system has non-trivial solutions only if the determinant of the matrix of coefficients is equal to zero, i.e. $|M - \lambda E| = 0$. The latter is called the *characteristic equation* of the matrix M. For our application of computing λ_2 values, M is replaced by the 3×3 matrix $S^2 + \Omega^2$. Accordingly, the resulting characteristic equation involves a cubic polynomial.

Solving cubic equations is a well-understood problem. For this work, we adopted a modification of *Cardan's solution* proposed in [55]. Assuming a cubic polynomial equation

$$ax^3 + bx^2 + cx + d = 0$$

the method distinguishes between whether the equation has three real roots (with two or three equal roots), three distinct real roots, or only one real root (with the

remaining roots being complex). Since $S^2 + \Omega^2$ is real and symmetric, we only have real roots and no complex roots at all. And this, in turn, means that just two cases have to be differentiated.

The actual differentiation is based upon the quantities y_N denoting the function value of the characteristic polynomial evaluated for the argument $x_N = -b/(3a)$ and for h being defined as $h = 2a\delta^3$ where $\delta^2 = (b^2 - 3ac)/(9a^2)$. If $y_N^2 = h^2$ there are three real roots (with two or three multiples) given by:

$$\begin{aligned} \alpha = \beta &= x_N + \delta \,, \\ \gamma &= x_N - 2\delta \,, \end{aligned}$$

where $\delta = \sqrt[3]{y_N/(2a)}$. The inequality $y_N^2 < h^2$ denotes the more frequent case of three distinct real roots. In this case, the eigenvalues are given by:

$$\begin{aligned} \alpha &= x_N + 2\delta\cos(\theta) \,, \\ \beta &= x_N + 2\delta\cos(2\pi/3 + \theta) \,, \\ \gamma &= x_N + 2\delta\cos(4\pi/3 + \theta) \,, \end{aligned}$$

where $\cos(3\theta) = -y_N/(2a\delta^3)$. The implementation of this solver algorithm using the cosine function provided with the C standard library is straightforward but it suffers from the computational costs of repeated trigonometric operations.

One common approach for alleviating this problem is to use tables storing—in our case—cosine values pre-computed for a small, fixed number of arguments. This technique can also be used very efficiently for speeding up the solver since unscrambling the above expressions yields that exactly three cosine values for the arguments $\arccos(-y_N/(2a\delta^3))/3 + 2k/3\pi$, $k \in \{0, 1, 2\}$, are needed. We, therefore, create a lookup table of three-component vectors storing pre-computed cosines for exactly these arguments and access the table with the inverse cosine argument translated to an integral table index. The three cosines can then be obtained simultaneously with a single table lookup and inverse cosine calculations during the λ_2 calculation become completely obsolete.

In fact, the results presented in [56] even suggest a solution which requires only a single cosine computation. However, this solution involves an additional square root which cannot be reasonably tabulated and, as before, it requires an address computation and a memory access for looking up cosine values; thus, the approach was not taken into consideration.

5.2.4 Evaluation

For comparison, λ_2 values were computed for the same data set defined on a regular grid using both PowerVIZ and the given software solution. Compared to

PowerVIZ, our implementation performs exceptionally well and shows a seven-fold speed-up. Nevertheless, even on high-performance computing hardware the λ_2 calculation performance is about 0.3 s per million cells (Pentium4 2.8 GHz)—too slow for interactive work considering data set sizes of several millions to tens of million grid cells and also considering that denoising and visualization are not yet included in these times.

5.3 GPU Implementation

The λ_2 criterion is highly suitable for a GPU implementation since it—as was argued in Section 2.6.2 during the discussion of general-purpose GPU computations—is a local algorithm solely based on the velocity gradient tensor and, thus, requires only a fixed number of input values. In this section we describe how the λ_2 algorithm can be mapped to programmable graphics hardware reusing the work of the previous section. The research results were first published in [92, 93, 94].

The hardware platform and graphics API used for the implementation are the ATI 9800 XT and DirectX/D3D with HLSL, respectively. This platform has a tight instruction limit of 64 + 32 = 96 instruction slots (arithmetic and texture instructions) but it does involve only a very minor performance penalty for multi-pass rendering, a technique indispensable for our application (see dashed path in Figure 2.12).

As for the software solution, the original velocity field is first inflated by adding border cells allowing for a uniform computation of partial derivatives. However, instead of storing the field in a 3D array the volume is cut into slices having constant Z-coordinates and saved in 2D RGBA floating point textures (the platform does not support floating-point RGB textures forcing us to leave the alpha component unused). For performance reasons, no 3D textures are used.

In contrast to the software solution which only computes the λ_2 scalar field, the GPU implementation also includes filtering and visualization functionality for completing the cycle defined in Section 5.1. All phases of the cycle are completely GPU-based and, once the vector field has been uploaded to GPU memory, at no instant require a read-back of any data.

5.3.1 Filtering

Noise is usually characterized by high-frequency components of a signal. Therefore, a low-pass filter must be used for denoising the data. Usually, the ability of a low-pass filter to suppress noise greatly depends on its support, i.e. the number of neighbors incorporated into the calculation of the filtered value. In a hardware implementation neighbors need to be determined by lookups into textures filled

by the application with the appropriate information. Thus, a filter of support N requires N^3 texture lookups for obtaining neighbor information. Our target platform only supports 32 texture lookups per pass; thus, for non-separable filters, i.e. filters that cannot be implemented as three 1D filters, the maximum filter support that can be implemented in a single pass is three. If kernels of wider support are required, separable filters and multi-pass rendering must be resorted to instead. For filtering vector field data, a filter should also be isotropic, i.e. rotation-invariant, since otherwise oriented features—like vortices—might be lost. Since the standard Gauss filter matches both criteria it was selected for this work.

When implementing a separable filter the order in which the 1D filters are applied in general is irrelevant. For an application storing the input data in a stack of Z-aligned slices, however, the complexity of the implementation and the amount of temporary texture memory are well affected.

If the data is filtered first in X- and Y-direction, only a single input and output texture are required for the first two passes since a single Z-aligned slice includes all the required input data and all output data again refers to a single slice. However, the final Z-direction rendering pass requires N slices to have already been filtered in X- and Y-direction; thus, $N + 1$ temporary floating point textures are required.

On the other hand, if the data is filtered first in Z-direction, N slices of the unfiltered vector data are used as input and the result is written to a single floating point texture. For the remaining passes, another floating point texture is required; thus, only two temporary textures are required (independent of the filter support) and no program logic is needed that caches the results of the XY-filtering for a final Z-filtering pass. Accordingly, for this work the latter approach has been used.

For the actual filtering, proxy geometry is then generated by rendering as many quadrilaterals as there are Z-aligned slices in the volume. Each quadrilateral, in turn, comprises as many pixels in the two dimensions as there are grid cells in X- and in Y-direction, respectively. For the X- and Y-directions, the filtering is done analogously—with the difference, however, being that a single texture has to be bound instead of N textures. In either case, for each generated fragment, a pixel shader issues the required number of texture lookups for determining neighbor information and subsequently computes the weighted sum. The (user-adjustable) Gaussian weighting factors required for applying the filter are pre-computed by the application and passed to the pixel shader as program parameters. Thus, no evaluations of exponential functions on a per-pixel basis are required.

The final filtering passes consume N texture lookups each and 32 and 12 instructions slots for the filtering in X-/Y- and Z-direction, respectively. Since, in either case, at most N texture lookups are required, filter supports of up to 31 can theoretically be implemented with the number of texture lookups restricted to 32. However, on our platform the number of texture samplers—a Direct3D construct

for configuring the texture access—is currently limited to 16 so the maximum filter support cannot exceed 15. For this work, a kernel size of 11 is used. The denoising capabilities of this filter for various variances of the Gauss function are illustrated in Figure 5.1, lower row.

5.3.2 Vortex Detection

A careful port of the optimized λ_2 algorithm devised in Section 5.2 requires almost twice as many instruction slots as are available on the ATI 9800. Hence it is impossible to implement the vortex detection in a single pass and the question arises where to cut the pixel shader into passes.

This split-up must not be arbitrary. When adjusting an algorithm to multi-pass rendering, intermediate results are written to a texture which in turn is made available to the subsequent pass. This means that the amount of intermediate data per pixel has to fit into an RBGA texel of at most 128 bit. If this does not suffice it is possible on some architectures to use multiple render targets (MRTs). Fortunately, the λ_2 method can very nicely be split into two passes.

The idea is to calculate the coefficients of the characteristic polynomial in the first pass and to solve the characteristic equation in a second pass. Since the characteristic polynomial is a cubic, only four floating point numbers need to be passed between the passes. This data tightly fits into an RGBA value and thus allows us to abandon MRTs.

As for the software implementation, the velocity gradient tensor can be efficiently computed by exploiting the vector operations provided by the graphics hardware. HLSL also already provides matrix manipulation functions; thus, one might be tempted to utilize these functions for setting up $S^2 + \Omega^2$. Unfortunately, the resulting code exceeds the instruction limit of 64 and, thus, must be discarded. In contrast, again taking into account that $S^2 + \Omega^2$ is symmetric and computing only the non-redundant elements, a valid single-pass shader is obtained requiring six texture lookups and 59 instruction slots—well within the limit of the hardware.

Once the sum matrix has been derived, the coefficients of the characteristic polynomial are calculated in a straightforward way and assigned to the RGBA components of the pixel color to make them available to the rendering pass solving the characteristic equation.

The solver pass also benefits from the optimizations devised for the software implementation. The HLSL instruction set includes both instructions for computing cosine and inverse cosine values. However, the `sincos` instruction (which the HLSL `cos` and `acos` instructions are mapped to) consumes eight instruction slots; thus, a total of 24 instruction slots are required for the three cosine calculations alone. As a result, when using the native instructions of the GPU, the maximum number of instructions is surpassed and the program is again rejected.

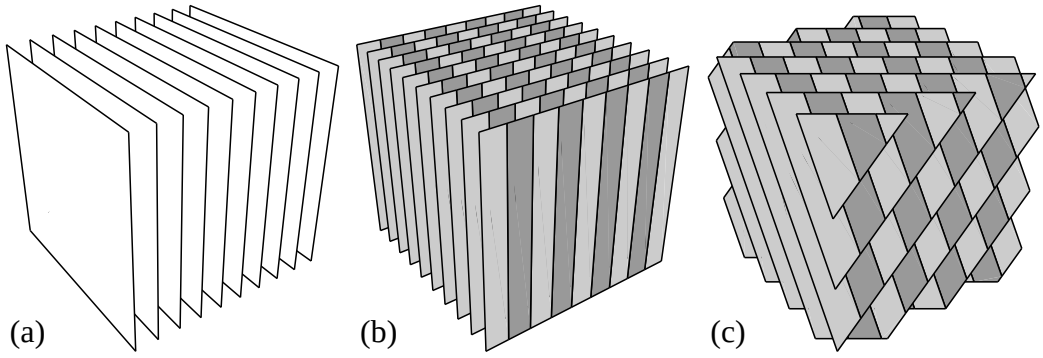

Figure 5.2 Comparison of slab filling methods. **(a)** Original slices, **(b)** pre-calculated axis-aligned slices, **(c)** viewport-aligned slices calculated on-the-fly.

Using a 1D RGBA texture of pre-computed cosine values in contrast requires only a single texture lookup for determining all three cosines, thereby dramatically affecting performance and the number of required instruction slots. Nevertheless, with two lookups and 55 slots the final implementation is close to the instruction limit, again illustrating the need for the optimizations introduced in Section 5.2.

5.3.3 Volume Visualization

The output of the λ_2 method is a scalar field; thus, in theory any of the volume visualization techniques presented in Sections 2.4.1 and 2.4.2 can be used for visualizing the sought-after vortex structures. However, while standard techniques for direct volume rendering require either 3D textures or three stacks of 2D textures, standard indirect volume visualization techniques require the scalar field to be located in main memory. Unfortunately, for technical reasons, in our GPU implementation the λ_2 data is written to a single stack of 2D textures. That is, applying either standard technique requires the scalar field to be transferred to main memory. Since this is a costly operation, we adopt an approach proposed in [72] which, in turn, originally bases on the work presented in [15].

The idea behind these approaches is to determine intersection polygons of viewport-aligned slices with the given stack of quadrilaterals and to interpolate color values on these polygons using the two neighboring textures. The original stack of quadrilaterals and the resulting intermediate slices are depicted in Figure 5.2 (a) and (c), respectively.

A drawback of this approach is the large number of intersection polygons that have to be calculated each time the volume is rotated. Our implementation, therefore, employs a slightly different approach which is a trade-off between speed and quality: Instead of determining viewport-aligned intersection polygons on-the-fly,

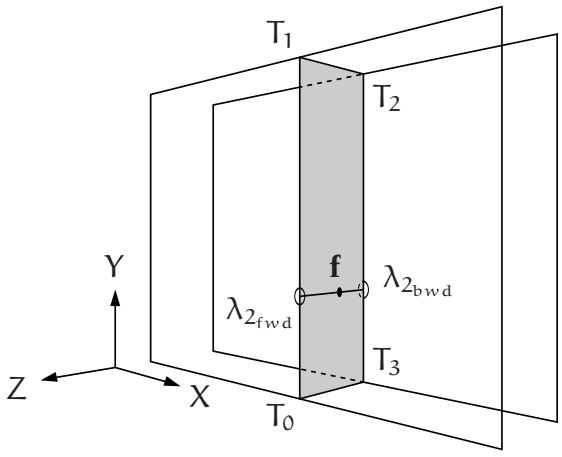

Figure 5.3 Interpolation of a single intermediate stripe from two axis-aligned Z-slices. The texture coordinates are $T_0 = (x, 0, 0)$, $T_1 = (x, 1, 0)$, $T_2 = (x, 1, 1)$, and $T_3 = (x, 0, 1)$, respectively. The values $\lambda_{2_{fwd}}$ and $\lambda_{2_{bwd}}$ are looked up using the first two components of the 3D texture coordinates interpolated for fragment **f**, the final λ_2 value at **f** is linearly interpolated from $\lambda_{2_{fwd}}$ and $\lambda_{2_{bwd}}$ using the third component.

we pre-compute sets of intersection polygons for the X- and Y-directions (both positive and negative), respectively, and switch between them (and the original stack) depending on the orientation of the volume bounding box to the viewer (Figure 5.2 (b)). The corresponding stripe interpolation in X-direction is illustrated in Figure 5.3.

This simplified approach not only enables us to pre-calculate the intersection polygons but also to send the geometry data of the slices only once to the graphics adapter as a vertex buffer. Since the slices of the original stack are equidistant (the grid is assumed to be regular), the amount of geometry stored in the vertex buffers can further be significantly reduced by storing only a single stack of stripes and rendering the remaining ones with a suitable translation applied. Thus, the approach is able to accelerate the visualization without taking a noteworthy amount of memory.

An isosurface—the most appropriate visualization for vortex structures—is efficiently extracted from the stack of slices by rendering the slices with all pixels within a user-defined interval around the isovalue set to opaque and the remaining pixels set to transparent. However, the resulting "surface" will have a uniform color which conceals its 3D structure. We have, therefore, integrated a lighting model into our system incorporating both ambient and diffuse parts [23]. Specular lighting was rejected due to high computational costs—we found that the specular component is about as costly as the ambient and diffuse terms taken together. Like

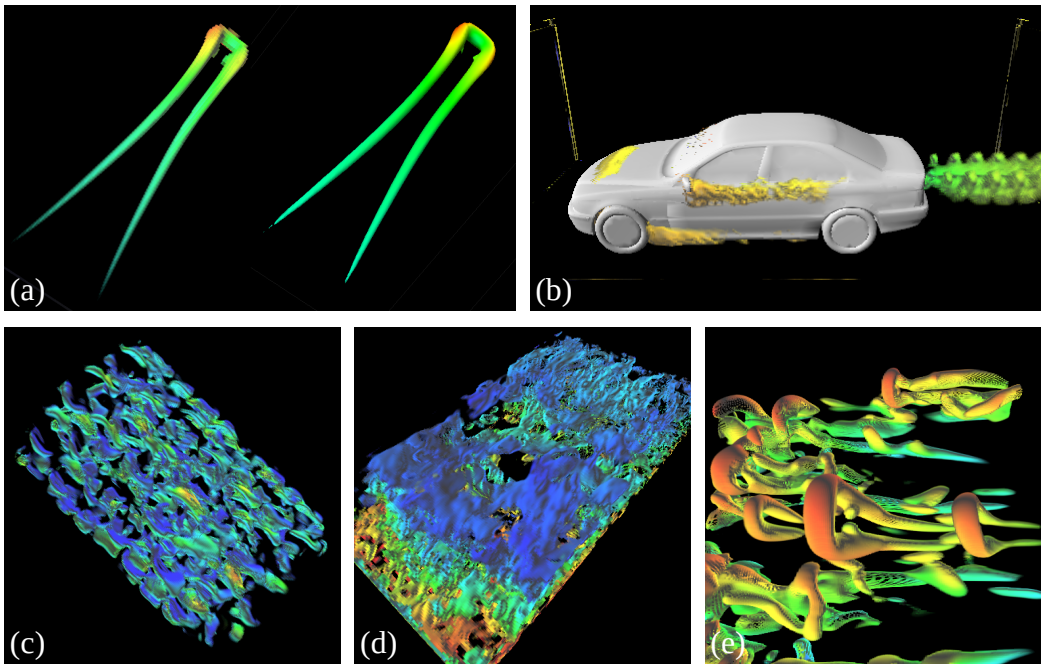

Figure 5.4 Vortices extracted and visualized with the GPU-based implementation of the λ_2 criterion. **(a)** Isolated Λ-vortex; **(b)** flow around car body (resampled grid); **(c)** flow through sphere filling; **(d)** measured water channel flow; **(e)** K-type transition experiments. Data sets courtesy IAG (University of Stuttgart), BMW AG, and LSTM (University of Erlangen).

the filtering, the lighting calculations are performed on a per-pixel basis in a pixel shader using the λ_2 gradient as surface normal [51]. This way, the rendering in Z-direction can be accomplished with one texture lookup and 15 arithmetic instruction slots, the rendering in X- and Y-direction with two texture lookups and 18 instruction slots.

Figures 5.1 and 5.4 show various visualizations generated with the GPU-based λ_2 implementation. The car body data set of Figure 5.4 (b) is the one already used in Figure 4.8. The water channel data (Figure 5.4 (d)) was obtained experimentally by LDA (Section 2.3.1), for the vehicle data set, a lattice-Boltzmann simulation (Section 4.1) was used (which, as discussed, is not vulnerable to noise and actually does not demand for interactive λ_2 computation). The remaining data sets were obtained by DNS.

5.3.4 Evaluation

General purpose computations on the GPU are different from software implementations in a number of ways. First, since GPU implementations store the input data in textures (which is a scarce resource, in current graphics adapters 256 MB), memory efficiency is a crucial topic. Second, the internal numerical accuracy of GPUs varies. On our platform, all computations are performed with 24 bit. Thus, numerically instable computations might give highly different results when performed on the GPU as compared to the CPU. Finally, the decision for porting an algorithm to graphics hardware is usually driven by performance considerations; that means, for a valid comparison the GPU implementation must be compared to an equally optimized software implementation.

Our GPU implementation stores the raw velocity data in 128 bit RGBA textures. These textures must not be modified since they are needed each time the filter characteristics are adjusted. To store the filtered results, another 128 bit floating point texture is required. This texture is reused for storing the gradients of the λ_2 scalar field required for lighting the isosurface. The remaining textures are independent of the size of the input data and of negligible size. Thus, if the input data comprises n grid cells, about $32n$ byte memory are consumed by the system. Thus, with 256 MB of texture memory data sets of about eight million grid points or equivalently a cube of the dimensions $200\times200\times200$ can be visualized. For most data sets obtained experimentally this will be sufficient. For many simulated flows, however, a bricking approach as proposed in [27] will be required.

The accuracy of the GPU implementation is illustrated in Figure 5.4 (a). The image shows an isolated Λ-vortex of $135\times225\times129$ grid cells visualized with the GPU-implementation (left) and PowerVIZ (right). In contrast to the GPU-approach which uses a direct volume rendering approach, PowerVIZ does not use direct volume rendering techniques for extracting isosurfaces but instead extracts a polygonal representation using a Marching Cubes approach (Section 2.4.2). As can be seen, the results are nevertheless very similar in general and only differ (visually) where the vortex tubes narrow and the number of slices over the profile is reduced.

The Λ-vortex data set was also used for evaluating the system performance. Our measurements show that the λ_2 computation on the chosen platform (ATI 9800 XT) is almost an order of magnitude faster than the optimized software implementation presented in Section 5.2. Accordingly, the performance gain with respect to PowerVIZ is about a factor of 60. Of all the phases, volume visualization in Z-direction uses the least processing time and runs at 26.3 fps, followed by the λ_2 computation (7.7 fps) and the filtering (Gaussian of kernel size 11) running at 6.8 fps. The most expensive phases are visualizations in X- and Y-direction in which case a large number of intermediate stripes needs to be rendered. For this

case the rendering performance drops to 4.7 fps.

Considering the complete cycle of filtering, vortex detection, and visualization we found that interactive work for data sets of up to six million grid cells is possible at about 2-3 fps. For practitioners analyzing noisy flow data, this presents a considerable improvement.

5.4 Visualization Techniques

The evaluation of the GPU-based λ_2 implementation given in Section 5.3.4 shows that of all the phases the rendering part is the most expensive. Optimizing this part thus promises the greatest performance gain. Redesigning the visualization part is also interesting from a visual quality point of view since the visualizations exhibit artifacts resulting from the interpolation of intermediate stripes (see vortex tubes in Figure 5.4 (e)). To keep the system all-GPU, however, we must confine ourselves to visualization techniques that are completely GPU-based and that do not require intervention by the CPU. And since vortex tubes are most naturally visualized by equi-valued surfaces, attention should be turned to isosurface visualization techniques.

For the following discussion we must differentiate between GPU-based *isosurface visualization* in which the surface is lost (as it is used in the original GPU-based λ_2 implementation by interpolating and blending intermediate stripes) and *isosurface extraction* which is mostly software-based but which results in an explicit geometrical representation allowing for further processing. For this work both directions were followed for improving the system—the latter with a view to improving the system performance, the former for improving the quality of the isosurfaces and to provide for advanced visualizations.

Unfortunately, however, no significant improvements in either direction currently seem attainable without the possibility of directly rendering into 3D textures (the lack of which already enforced the rather complicated volume rendering approach of the previous section). Thus, neither of the techniques described in the following at the time can be integrated into the λ_2 implementation. We will, therefore, discuss these visualization techniques because of their relevance as technology demonstrations and their contributions to general scientific volume rendering rather than their impact on the acceleration of numerical flow visualization.

5.4.1 GPU-Based Extraction of Isosurface Geometry

In contrast to GPU-based isosurface *visualization*, using graphics hardware for *extracting* isosurface geometry is sparsely researched and only addressed in a single publication [64] discussing the use of vertex processors for implementing the

Marching Tetrahedra algorithm [14]. This algorithm is conceptually very similar to the Marching Cubes approach introduced in Section 2.4.2 in that it compiles the final isosurface geometry from a number of subsurfaces independently computed for the cells comprising the grid. However, while the primitive for which isosurface intersections are computed are cuboids in the Marching Cubes algorithm, the Marching Tetrahedra algorithm uses tetrahedra. This modification is trivial but it results in a number of benefits since, first, ambiguities are eliminated and, second, because isosurfaces can now be extracted from unstructured grids, too (due to the fact that any polyhedron can be subdivided in tetrahedra, Section 4.3.1). When applied to regular grids, however, the Marching Tetrahedra approach proves disadvantageous with respect to computational costs since—depending on the tetrahedral subdivision scheme—the number of primitives that must be processed increases by a factor of five or six.

The main drawback of the GPU-based Marching Tetrahedra implementation given in [64] is the utilization of the vertex processing unit since, as indicated in Section 2.6.2, this unit is less powerful than fragment processors. In the following, we thus present the first fragment-program–based isosurface extraction which, for simplicity, also implements the Marching Tetrahedra approach rather than the Marching Cubes algorithm. This research was published with T. Klein [40].

The system is implemented as two rendering passes. For the first pass, a quadrilateral is rendered generating four fragments per tetrahedron and the intersection polygons—or rather: the edge intersections comprising the polygons—computed on a per-fragment basis and written to an on-board graphics memory buffer. In the second pass, this buffer is bound as vertex array and the geometry rendered. The tetrahedral grid is encoded in textures in indexed representation which means a tetrahedron is defined by a set of four vertex indices each of which, in turn, can be uniquely mapped to 3D vertex coordinates. This choice is easily motivated when considering the memory consumption of a grid comprising V vertices and T tetrahedra. With direct encoding each tetrahedron consumes $3\times4 = 12$ memory units, each unit being able to store a single coordinate component. Thus, the overall memory consumption is $M_d = 12T$ units. On the other hand, when using an indexed representation the overall consumption is $M_i = 3V + 4T$ units under the assumption that indices are the same size as coordinate components. On average, however, we found that $T/V \approx 5 - 6$. Thus, substituting $T = 5V$ yields $M_d = 60V$ and $M_i = 23V$. This result clearly shows that on average indexed encoding is able to store more than two times as many tetrahedra as direct encoding—a benefit not to be undervalued given the limited amount of texture memory provided by graphics cards.

For the implementation the same graphics cards series as for the GPU-based λ_2 (Section 5.3) implementation was used; thus, again the maximum number of instruction slots is limited to 64, the available texture memory 256 MB. In ad-

dition, the design is influenced by the maximum number of texture indirections, i.e. the maximum length of the texture dependency chain where data looked up in a texture is used for computing a second pair of texture coordinates. On the chosen platform, only four indirections are supported. OpenGL was used as graphics API.

System Architecture

When intersecting a tetrahedron, a plane may cut off up to three vertices. Cutting off one vertex obviously is identical to cutting off three vertices since in either case there are three vertices on one side of the isosurface while one vertex is located on the other side. Similarly, when cutting off two vertices there are always exactly two vertices on either side. That is, only the three vertex ratios (inside to outside or vice versa) 0:4, 1:3, and 2:2 have to be handled. For the first case, no polygon must be generated. For the second case, a triangle must be rasterized. And for the third case, a quadrilateral must be drawn. In OpenGL, when drawing a quadrilateral, vertices are allowed to coincide. Thus, the drawing of intersection polygons can be handled uniformly by drawing a quadrilateral independent of the intersection case and, if necessary, by replicating vertices. This, in turn, means that the implementation can be simplified by always generating exactly four fragments per tetrahedron, each of which determines an intersection vertex. Deciding which fragment is responsible for which edge intersection poses the main difficulty in mapping the Marching Tetrahedra algorithm to graphics hardware.

Initially, all information available for a single fragment are its position (x, y) in window coordinates. Since four consecutive fragments refer to the same tetrahedron a way is needed for computing a common tetrahedron index τ for each of the fragments of the group of four. In our implementation we use a two-component index $\tau = (\lfloor x/4 \rfloor, y)$. By appropriately scaling τ we can obtain valid texture coordinates for accessing a 32-bit 2D RGBA texture storing the scalars corresponding to the tetrahedron vertices (Figure 5.5). The result of the texture lookup is a vector $\mathbf{S}$ of four scalars. By replicating the user-specified isovalue to a four-component vector $\mathbf{C}$, a bitmask β then can be determined by a single vector-operation informing about whether the vertex scalars are above or below the isovalue: $\beta = \mathbf{S} < \mathbf{C}$. Since the bitmask has four bits, there are 2^4 = 16 different cases that have to be differentiated. As for the Marching Cubes algorithm, however, we can exploit symmetries to simplify the differentiation.

Two cases are trivial, the first being the case where all bits are cleared, the second being the case with all bitmask bits set. In both of these cases there is no intersection. For the remaining 14 cases we make the following observations. If the isosurface cuts off a single vertex being the only of the group of four having a scalar value *less* than the isovalue, then a single triangle is generated. Obviously,

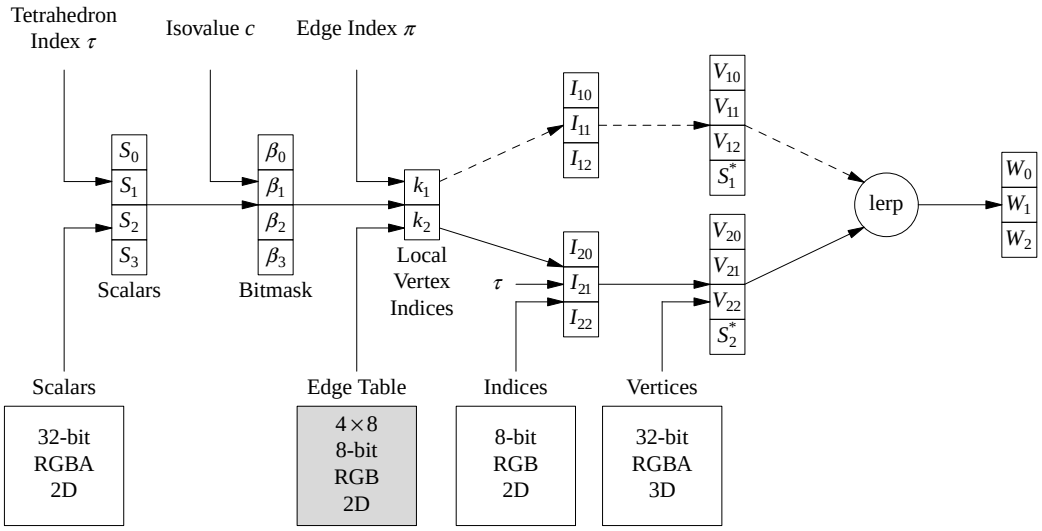

Figure 5.5 Overview of GPU-based system for extracting isosurface geometry. The shown path computes the intersection of a single tetrahedron edge with the isosurface. The shaded box indicates a texture lookup replaced by fragment shader computations.

the identical triangle would be generated if the outlier were the only one having a scalar value *greater* than the isovalue. With respect to the bitmask β this means that the two cases for a given bitmask and its bitwise complement actually present a single case. In fact, we already used this for the trivial cases; thus, there remain exactly seven cases as depicted in Figure 5.6.

In order to exploit symmetries we need to invert the bitmask for half of the original 16 cases. And as will become clear below, we further demand that after the reduction, the number of bits set in the bitmask for the non-trivial cases informs about the type of polygon that must be generated for the respective case. Intuitively, we define a single set bit to indicate a triangular intersection polygon and two set bits to indicate the quadrilateral case. This results in the configurations shown in Figure 5.6. The upper bitmask of each case shows the bitmask the case is reduced to.

For deriving the criterion for inverting the bitmask, we make two further observations. The first observation is that all but a single bitmask have—when interpreted as a decimal number—a value greater than eight (the outlier has value seven). The second observation is that when adding to the decimal interpretation the number of bits set in the bitmask, the resulting value is always greater than nine for bitmasks to be inverted and less than or equal to nine for the remaining cases. Thus, a suitable and efficient criterion is $(8,4,2,1)^\mathsf{T} \cdot \boldsymbol{\beta} + (1,1,1,1)^\mathsf{T} \cdot \boldsymbol{\beta} > 9$

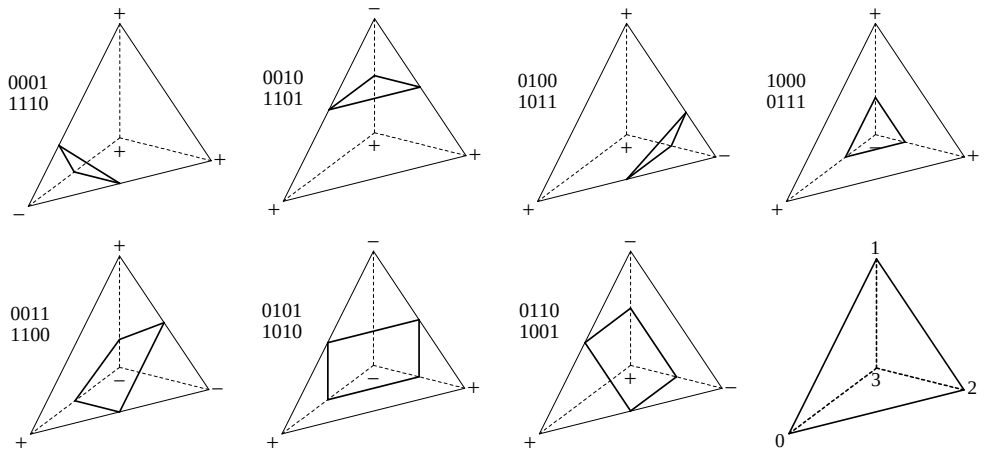

Figure 5.6 Marching Tetrahedra cases. Cases resulting in triangles are shown in the upper row, quadrilateral cases in the lower row. Trivial cases where the isosurface does not intersect the tetrahedron are not shown. Bit positions associated with a vertex are shown in the lower right.

or, equivalently, $(9, 5, 3, 2)^T \cdot \beta > 9$.

Using the (adjusted) bitmask determining the type of intersection polygon is simple because the bitmasks were chosen such that a single bit indicates a triangle while two bits indicate a quadrilateral. In contrast, determining edges whose isosurface intersections define the polygon vertices is complex and usually table-driven. On the chosen graphics hardware, using tables results in an excess in the number of texture indirections and, thus, cannot be used. Accordingly, some algorithm must be derived for emulating a lookup table.

We will discuss the triangle case first. Rendering a triangle by means of a quadrilateral requires one vertex to be duplicated, i.e. we need to reduce the four fragments to three cases. Each case defines an edge intersection, i.e. we need to compute four edge intersections with one edge intersection replicated. We enumerate the tetrahedron edges and introduce edge indices $\pi = x \bmod 4$ indicating which tetrahedron edge a fragment must process. Obviously, all edges to be intersected share a single common vertex. We can determine the local (i.e. with respect to the tetrahedron) index k_1 of this vertex by $k_1 = (0, 1, 2, 3)^T \cdot \beta$. To determine the indices of the vertices defining the edge end points we need the four-to-three case reduction which is accomplished by $k_2 = (k_1 + \pi \bmod 3 + 1) \bmod 4$. This replicates the last vertex, i.e. the fragments with $\pi = 2$ and $\pi = 3$ redundantly compute the same intersection.

The quadrilateral case is more complex. Again, we make two observations: First, the isosurface intersects four of the six tetrahedron edges and, second, it intersects all four tetrahedron faces. That means two edges $e_1 = (i_0, i_1)$ and

$e_2 = (j_0, j_1)$ are *not* intersected and these edges do *not* share a common vertex. We can prove this fundamental property by assuming the edges had a common vertex. If this were the case, then the two edges would define a tetrahedron face. Since all tetrahedron faces are intersected by the isosurface (observation #2), this face would be intersected, too. However, as e_1 and e_2 do not intersect the isosurface (observation #1) the face cannot be intersected by the isosurface—a contraction that can only be resolved if the original assumption (e_1 and e_2 sharing a vertex) was wrong. The intersected edges are thus given by the ordered pairs of the cross product $\{i_0, i_1\} \times \{j_0, j_1\}$.

Considering that $\{i_0, i_1, j_0, j_1\}$ is the set of tetrahedron vertices, it is seen that one vertex may be chosen freely (and this vertex can be chosen identical for each case). To determine the remaining vertex indices we again use efficient dot products with the bitmask $\boldsymbol{\beta}$ and vectors $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$, respectively. Each of these vectors has four components and from the edge table (as it would be used for a software solution) it is known what local vertex index the dot products should be equal to when multiplying $\mathbf{b}_i$ with the classification bitmask $\boldsymbol{\beta}$. That is, we can set up systems of linear equations and solve these for the sought-after vector components b_{ij}. Thus, e.g., fixing $j_1 = 3$ we obtain for $\mathbf{b}_3$ from Figure 5.6, lower row, the conditions $(0,0,1,1)^T \cdot \mathbf{b}_3 = 2$, $(0,1,0,1)^T \cdot \mathbf{b}_3 = 1$, and $(0,1,1,0)^T \cdot \mathbf{b}_3 = 0$. Solving the resulting linear system of equations

$$\begin{array}{ccccccc} & & b_{33} & + & b_{34} & = & 2 \\ b_{32} & & & + & b_{34} & = & 1 \\ b_{32} & + & b_{33} & & & = & 0 \end{array}$$

yields $\mathbf{b}_3 = (0, -1/2, 1/2, 3/2)^T$. The vectors $\mathbf{b}_1$ and $\mathbf{b}_2$ are obtained analogously. Thus, the non-intersected edges $e_1 = (i_0, i_1)$ and $e_2 = (j_0, j_1)$ are then given by

$$i_0 = \mathbf{b}_1 \cdot \boldsymbol{\beta} = \begin{pmatrix} 0 \\ 1/2 \\ 1/2 \\ -1/2 \end{pmatrix} \cdot \boldsymbol{\beta} \ , \quad i_1 = \mathbf{b}_2 \cdot \boldsymbol{\beta} = \begin{pmatrix} 0 \\ 3/2 \\ 1/2 \\ 1/2 \end{pmatrix} \cdot \boldsymbol{\beta}$$

and

$$j_0 = \mathbf{b}_3 \cdot \boldsymbol{\beta} = \begin{pmatrix} 0 \\ -1/2 \\ 1/2 \\ 3/2 \end{pmatrix} \cdot \boldsymbol{\beta} \ , \quad j_1 = 3 \ ,$$

respectively. The dot products, however, only return the indices i_0, i_1, j_0, and j_1 but they do not yield the pairs of indices for a given edge index π. Thus, we

derive another vector $\boldsymbol{\epsilon}(\pi)$ which, when multiplied by $\boldsymbol{\Lambda} = (i_0, i_0, i_1, i_1)^T$ and $\boldsymbol{\Gamma} = (j_0, j_1, j_1, j_0)^T$ returns the respective indices of the edge end points. In our implementation $\boldsymbol{\epsilon}(\pi)$ is constructed as a vector of four interpolating polynomials which are constructed such that for any edge index π exactly one component is set to one—the one required for selecting any of the indices i_0, i_1, j_0, j_1—while the remaining components are set to zero:

$$\epsilon(\pi) = \begin{pmatrix} \frac{1}{6}\,\pi\,(\pi-1)\,(\pi-2) \\ -\frac{1}{2}\,\pi\,(\pi-1)\,(\pi-3) \\ \frac{1}{2}\,\pi\,(\pi-2)\,(\pi-3) \\ -\frac{1}{6}\,(\pi-1)\,(\pi-2)\,(\pi-3) \end{pmatrix}.$$

The index k_1 of the first edge end point is then given by $k_1 = \boldsymbol{\epsilon}(\pi) \cdot \boldsymbol{\Lambda}$, the index k_2 of the second end point by $k_2 = \boldsymbol{\epsilon}(\pi) \cdot \boldsymbol{\Gamma}$.

To stay within the instruction limit, a construction must be used that computes both the index pair required for the triangular case and the indices of the quadrilateral case. Also, if there is no intersection at all, all index pairs must be identical. The former again is accomplished by considering the number of bits $n = (1,1,1,1)^T \cdot \boldsymbol{\beta}$ set in the bitmask. For the latter, we compute a factor $f = \max(0, \min(1, n))$ which becomes zero for the trivial case and one for the non-trivial case.

The final step in the first rendering pass then involves determining the actual vertex coordinates. These coordinates are obtained in a two-step process due to the indexed representation of the tetrahedral grid (Figure 5.5). First, the global vertex indices $\mathbf{I}_1$ and $\mathbf{I}_2$ of the vertices comprising the intersected edge are determined from a 2D RGB texture using τ and the computed offsets k_1 and k_2 as texture coordinates. These triplets, in turn, are then used as texture coordinates for accessing the actual vertex coordinates $\mathbf{V}_1$ and $\mathbf{V}_2$ stored in a 3D RGBA texture. Obviously, a three-component RGB texture would suffice for the latter. However, for linearly interpolating the intersection vertex $\boldsymbol{W}$, interpolation weights and, accordingly, the scalar values associated with a vertex are required. Extracting the correct scalars from the vector $\boldsymbol{S}$, however, again involves complex element selection; thus, we (redundantly) store the scalar field in the alpha component of the 3D RGBA texture holding the vertices. That is, the required scalars are readily available once the edge vertices have been determined. Finally, the interpolated vertex coordinates $\boldsymbol{W}$ are multiplied by the factor f, thereby mapping all output vertices to the origin when no intersection is found.

Surface Geometry Rendering

The ideas of the previous section can be employed to render the intersection vertices to an off-screen buffer, then to transfer them to main memory, and finally to

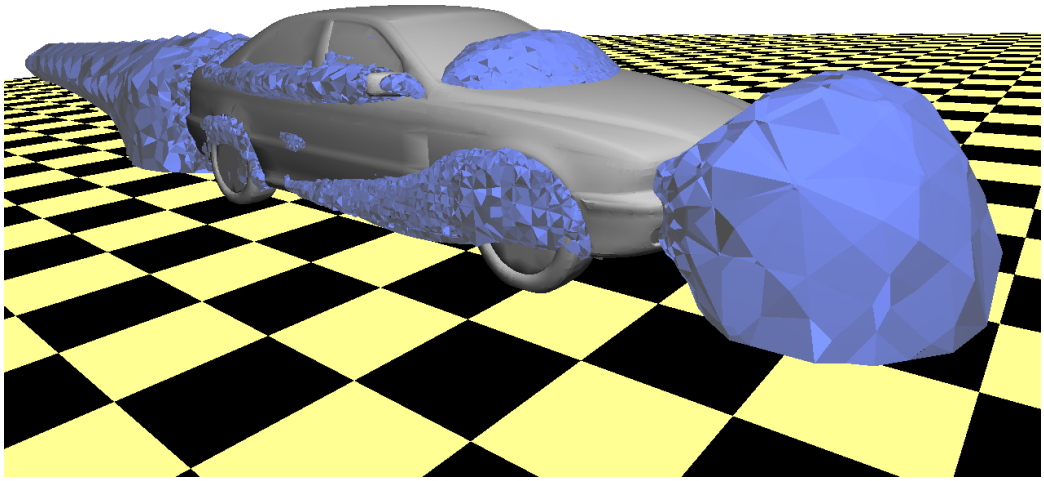

Figure 5.7 Isosurface geometry extracted on the GPU. The scalar property is velocity magnitude, the data is defined on an unstructured grid. Data set courtesy BMW AG.

render the polygonal isosurface representation by re-injecting the vertex data into the graphics pipeline. However, read-backs from graphics memory are expensive operations and should be avoided.

The (unofficial) ATI superbuffer extension allows for a more efficient solution [50]. It introduces generic graphics memory objects which we can render to and which can be bound as texture objects and vertex arrays for subsequent rendering passes. Superbuffers thus can be considered an advanced support for multi-pass rendering used in Section 5.3 and illustrated in Figure 2.12.

Using the superbuffer functionality, rendering the isosurface geometry extracted in the first rendering pass becomes straightforward. However, due to the tight instruction limit of our platform, computing surface normals for pleasing visualizations is impossible. Thus, to each generated intersection vertex we attach the index of the tetrahedron the vertex belongs to and—when rendering the surface geometry—use the index for looking up gradient information pre-computed per tetrahedron and made available in an additional texture map. Obviously, computing a single gradient per tetrahedron results in the same normal being assigned to all vertices comprising the intersection polygon. Thus, only flat-shading is supported, yielding visible color discontinuities across polygon boundaries. Figure 5.7 gives an example of an isosurface extracted and visualized with our GPU-based Marching Tetrahedra implementation. The visualized vehicle data set is the same unstructured grid already used at numerous places in the preceding chapters (Sections 4.3.2 and 5.3.3).

Evaluation

Due to multiple rasterization units available on modern graphics cards, GPU-based isosurface extraction results in a massively parallel isosurface extraction. In addition, since the superbuffer extension supersedes bus transfers, the extracted polygonal surface representation can be rendered directly, thereby circumventing communication bottlenecks. These facts argue for a high extraction performance. In fact, measurements show that the given implementation is superior to any GPU-based implementation described previously [40].

In practice, however, the system disappoints in two respects. First, due to flat-shading the visual quality of the visualized isosurfaces is unsatisfying. And second, despite a peak performance of about 7.3 million tetrahedra/second the system is almost an order of magnitude slower than comparable software implementations for actual isosurface computations. The reasons for the latter are manifold.

First, isosurfaces are an indirect volume visualization technique, meaning only a subset of the voxels comprising the volume contribute to the final visualization. Therefore, having to transfer the entire volume to the GPU for extracting the isosurface is necessarily wasteful. In principle, this also applies to the GPU-based λ_2 implementation. In this case, however, the scalar field is *not* stored in main memory which means in order to use a software implementation an expensive memory read-back again will be required. Directly visualizing the vortices with a slice-based volume rendering rendering approach is thus advantageous.

A second reason for the low performance of the GPU-based isosurface extractor is lack of optimizations; thus, e.g., while in a software implementation tetrahedra which are not intersected by the isosurface are simply skipped, the GPU solution processes these cases, too, and merely skips the rendering by making sure the respective vertices of the quadrilateral coincide. Interval trees as proposed in [10] might be able to alleviate this problem. Using these acceleration structures, however, again requires a read-back to main memory; thus, whether GPU-based isosurface extraction eventually will be beneficial for the application of flow visualization once the technological preliminaries are met remains to be seen.

5.4.2 GPU-Based Volume Ray-Casting

The GPU-based isosurface extraction algorithm described in the previous section predominantly aimed at improving the isosurface rendering performance. Another aspect of improving the GPU-based λ_2 computation is to improve visual quality since slice-based/texture-based volume rendering is doomed to moderate image quality since, first, with planar proxy geometry the sampling distances are non-uniform (Figure 2.11) and, second, because framebuffer blending operations have

a low accuracy compared to the accuracy graphics cards supply for general numerical computations (16 bit vs. 24 or 32 bit). As seen in Section 2.4.1 this blending is an efficient way for approximating the volume rendering integral. However, numerical integration by blending operations is also a kind of implicit programming which makes it hard to modify the original volume rendering algorithm and to integrate optimizations; thus, slice-based volume rendering is also inflexible.

In contrast, ray-casting as introduced in Section 2.4.1 does not suffer from non-constant sampling rates. Furthermore, on graphics cards supporting the functionality of Shader Model 3 [53] or the comparable `ARB_fragment_program2` [58] extension of OpenGL which extend programmable graphics hardware about dynamic looping and branching and which vastly increase the maximum number of instruction slots and texture accesses, single-pass ray-casting can be implemented almost as easily as on a CPU. That is, the numerical integration for evaluating the volume rendering integral is implemented explicitly which renders GPU-based ray-casting very flexible. And as a result, GPU-based ray-casting also remains unaffected by inaccurate framebuffer blending operations since no blending functionality is utilized at all.

Numerous GPU-based implementations of the ray-casting algorithm for both structured and unstructured grids have been presented [42, 79, 106]. Common to these implementations, however, is that they perform multiple rendering passes. Single-pass GPU-based ray-casting was first presented in [59]. The described implementation, however, must be considered a design study rather than a productive system because it is limited in both functionality (direct volume rendering) and visual quality. The latter is a result of a sampling algorithm sampling the volume at a constant number of positions independent of whether the viewing ray traverses the whole volume or only a minor part of it. Obviously, the benefit of constant sampling rates inherent in ray-casting is thus lost—and even worse, it is lost at the costs of suboptimal performance.

In this section we present a flexible system for ray-casting in graphics hardware that does not have these drawbacks and which exhibits all the benefits of ray-casting outlined above. The work was first published in cooperation with M. Strengert and T. Klein [99].

Framework

Our system consists of two parts: a set of shaders each implementing a certain visualization technique and the actual OpenGL-based framework coordinating the rendering process. We will discuss the framework first.

Like any volume renderer the framework provides some standard functionality like handling user interactions, providing transfer function functionality, and functionality to load the volume data and to initialize volume textures. We assume

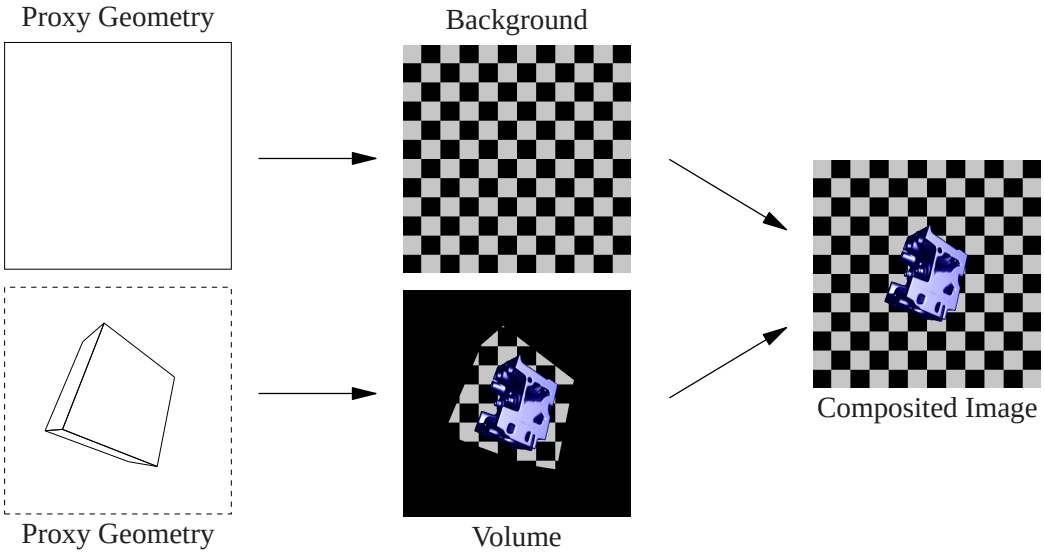

Figure 5.8 The two-step rendering process of the GPU-based ray-casting implementation. In the first step, the background is rendered by drawing a viewport-filling quadrilateral (top row), in the second step the volume visualization is added which seamlessly overdraws certain parts of the background (bottom row).

the volume data to be in unsigned character format and defined on a regular grid. Thus, there is a certain number of slices S_c in each dimension c, $c \in \{x, y, z\}$, and a corresponding slice thickness D_c (Section 2.2.1).

The scalar data defining the volume is stored in the alpha component of a 3D power-of-two RGBA texture with 8 bit per component. Considering the number of slices S_c, the texture dimensions are thus given by $T_c = 2^n \geq S_c$ where $\forall m < n : 2^m < S_c$. In the remaining three color components we store precomputed gradients which, as in Section 5.3.3, are used for surface lighting and which are determined optionally by central differences (Section 2.2.1) or a more advanced $3\times3\times3$ Sobel operator [88].

As for any fragment-program–based application some proxy geometry must be rendered to generate fragments. For this purpose, we render the volume bounding box which we define to be axis-aligned and to have one corner at the origin and the opposite vertex coordinates set to the normalized bounding box extents (E_x, E_y, E_z) which are formally given by $E_c = e_c/\max(\{e_x, e_y, e_z\})$ where, in turn, e_c denotes the actual volume extents $e_c = S_c \cdot D_c$. Of course, fragments need only be generated for ray entry positions but not for exit positions. Thus, back-face culling is enabled for removing those bounding box sides facing away from the camera and to make sure that a minimum total number of fragments is

generated, one fragment for each ray that needs to be cast into the volume[1]. To get a decent view of the volume the modelview matrix M is further set to translate the volume center $\mathbf{C}$ to the origin and to take into account any user-specified transformations.

Generating fragments is not sufficient for setting up rays since the fragments have not been assigned world coordinates which—in conjunction with the camera position—allow for computing the ray direction and which provide the ray starting positions. Therefore, texture coordinates are assigned to the bounding box vertices which are identical to the corresponding vertex coordinates. Texture coordinates, however, are unaffected by the modelview matrix M and, thus, would need to be transformed manually to account for changed viewing parameters set by the user. Considering, however, that for each sampling position we need to check whether the volume is left and, hence, whether the sampling must terminate it is seen that performing this test is significantly easier to do if the volume remains axis-aligned. As will be shown below in the former case the test basically reduces to two vectors comparisons—one against the origin, one against the upper bounding box boundary—while in the latter case expensive ray/box intersections are required.

Rendering the front faces of the volume bounding box is advantageous with regard to the number of generated fragments but it obviously results in a viewport being filled only partially. If uniform backgrounds are desired—as it will usually be the case for numerical flow visualization—this poses no problem since the missing pixel information can be recovered by simply drawing a viewport-filling quadrilateral before rendering the volume. This approach can also be used for patterned backgrounds by storing the background image in a texture and by adding an additional texture lookup to the fragment program. A more flexible solution, however, allows for arbitrary background patterns and does *not* demand the background pixel coordinates to match the window coordinates of the fragment. While this does not necessarily apply to flow visualization, as will be shown below there are some general volume visualization techniques actually requiring this functionality. Thus, for the moment we assume that the fragment program implementing the ray-casting is capable of determining the background pattern. Then this functionality will obviously be available for any world coordinate position and not

[1]It should be noted that this approach suffers from clipping problems as the camera and, accordingly, the near clipping plane approach the bounding box since no fragments are generated for rays starting inside the volume. Using the OpenGL stencil test [109] this drawback can be fixed as follows: 1. Render the back faces only and set the stencil value to one for the respective fragments. 2. Render the front faces only and set the respective stencil values to two. For each generated fragment a ray is cast into the volume as described above. 3. Render the near clipping plane where the stencil value equals one. This generates fragments for starting positions inside the volume.

```
I ← isovalue
X ← ray entry position
P ← volume scalar at X

Set up parametric equation of sampling ray
while inside volume bounding box do
    S ← volume scalar at current position
    if (S − I) · (P − I) < 0 do
        X' ← linearly interpolated isosurface intersection
        G ← re-oriented gradient at position X'
        Perform lighting calculation using G
        Exit loop
    end if
    Advance along ray
    P ← S
end while

Determine background pixel
Blend background
```

Figure 5.9 Algorithm for GPU-based isosurface extraction using ray-casting.

only those used for starting rays. That is if we are able to generate fragments with valid world coordinates for any viewport pixel outside the projected bounding box, then we can reuse the ray-casting shader for rendering the background, too. To derive the necessary proxy geometry we unproject (`gluUnProject(3G)`) the four viewport corners using the modelview matrix and the projection matrix and subsequently render a viewport-filling quadrilateral using the unprojected vertices for specifying both vertex and texture coordinates. Thus, although the actual volume ray-casting comprises a single pass, the rendering process coordinated by the framework consists of two phases, the background rendering and the volume ray-casting. The process is illustrated in Figure 5.8.

Single-Pass Volume Shader

The most important volume visualization technique for this work are isosurfaces. Figure 5.9 gives the basic algorithm for isosurface extraction using GPU-based ray-casting as implemented in our system.

We start by computing the position where the ray enters the volume. In our case this is trivial since the entry positions are readily given by the texture coor-

dinates which, in turn, are bilinearly interpolated automatically for each fragment by the GPU. For setting up the parametric ray equations in the next step we require another point—the camera position—which as mentioned above needs to account for the transformations dictated by the modelview matrix M. Similar to Section 5.3 we again want to keep the fragment program as compact as possible and, thus, we do not just perform a matrix/vector multiplication. Rather we take into account that in OpenGL the camera position is fixed to the origin and that, as seen in Section 3.5 for computing stereo views, instead of moving the camera the scene is moved. That is, in homogenous coordinates the camera is given by $(0,0,0,1)^T$ and this we want to transform with the inverse modelview matrix M^{-1}. However, with all components except for the homogenous extension being zero the result of the multiplication only extracts the translational part of the matrix stored in the 4th column. And this of course can be determined in a much more efficient way, not just by reading the column (this is not supported by `ARB_fragment_program2`) but by first transposing the matrix and then reading the respective row. The ray equation then is set up in a straightforward way (`DP3` implements a three-component inner product, `RSQ` is a reciprocal square root operation):

```
# Compute the camera position
MOV camera, state.matrix.modelview.invtrans.row[3];

# Compute the ray direction
SUB geomDir, geomPos, camera;

# Normalize the direction
DP3 geomDir.w, geomDir, geomDir;
RSQ geomDir.w, geomDir.w;
MUL geomDir, geomDir, geomDir.w;
MOV geomDir.w, 0.0;
```

Once the ray has been set up, a loop is entered for sampling the volume. Since `ARB_fragment_program2` does only allow for a maximum of 255 iterations per loop, this sampling is accomplished with a set of two nested loops. The iteration count for these loops is passed in from the application rather than being set to the iteration maximum. Although the latter would be possible due to the existence of `BRK` instructions—the equivalent of the C language `break`—this solution is discarded since passing the iteration count as program parameter allows for differentiating between the rendering phases solely based on the iteration count. That is, for rendering the background a zero iteration count is passed (thereby completely skipping the volume sampling part), for rendering the volume, the maximum iteration count is passed. In either case, the identical fragment program can be used.

For detecting isosurface intersections we compute both the difference between the isovalue and the scalar value at the previous sampling position, and the difference between the isovalue and the current scalar value. Obviously, differing signs indicate an intersection. This approach is robust in that it works independently of the sampling distance; however, for large sampling distances the uncertainty about where the isosurface is actually intersected becomes considerable. We thus assume a linear gradient of the scalar field between the sampling positions and linearly interpolate a guess for the ray/isosurface intersection. As for the GPU-based λ_2 implementation, a lighting calculation is then performed for attaining pleasing visualizations. Again, pre-computed gradient information is used for approximating the surface normal. The set of loops is then exited.

If no isosurface intersection is found, we advance along the sampling ray to obtain a new sampling position. For this position we need to determine whether it lies inside the volume bounding box or whether it lies outside of it. As mentioned above, since the ray entry positions are not transformed, this test is efficiently done with two vector comparisons comparing against the origin and the maximum texture coordinates:

```
# Compare to min/max
SGE temp1, pos, 0.0;
SLE temp2, pos, texMax;
DP3 temp.r, temp1, temp2;
SEQC temp.r, temp.r, 3.0;

# Outside volume, skip the rest
BRK (EQ.x);
```

The two instructions `SGE` and `SLE` perform component-wise comparisons for "greater or equal" (`SGE`) and "less or equal" (`SLE`), respectively, and set the result to zero if the condition is *not* met and to one if the condition is met. Thus, obviously, the dot product of the result vectors is equal to three if and only if all conditions are met or, equivalently, if the sampling position is inside the volume.

It should be noted that the given code sample uses the maximum texture coordinates `texMax` as upper boundary and *not* the normalized volume extents (E_x, E_y, E_z) of the coordinate system in which the ray starting positions are specified. This illustrates that the sampling positions are transformed to texture space prior to performing the test. For efficiency, this transformation is done outside the loops by applying the scaling—with the factors $F_c = \max(\{e_x, e_y, e_z\})/(T_c \cdot D_c)$ using the above notation—to the entry positions and the ray directions instead of to the individual sampling positions. For the inside test, transitioning to texture space clearly poses no advantage. However, since the untransformed sampling

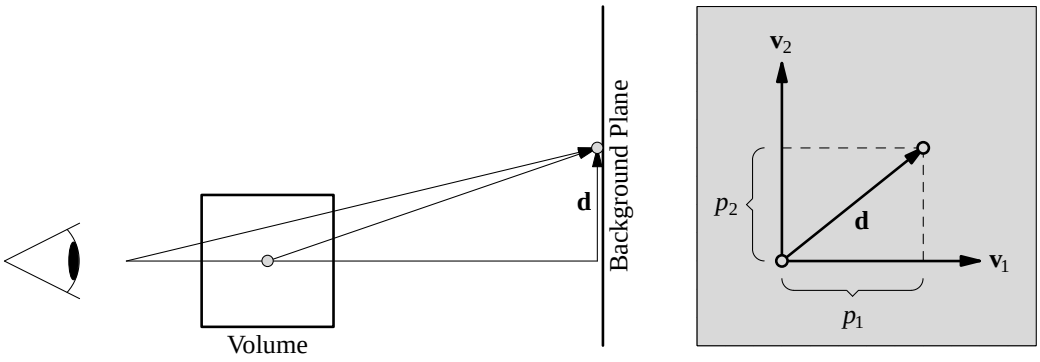

Figure 5.10 Determining the texture coordinates for accessing the texture storing the background image. Left: Side view, right: seen from the eye.

position cannot be used as 3D texture coordinates for accessing gradients and volume scalars, this step is inevitable.

Independent of whether the isosurface is hit or missed for a given fragment, as soon as the pair of loops is terminated a (preliminary) fragment color is known. In the latter case, the alpha value is set to zero, meaning the color is transparent. In the former case, the color is opaque (the standard visualization of isosurfaces). Thus, assuming the background pixel is known, a blending operation can then be used for the final compositing.

For determining the background pixel we start by defining a plane whose normal matches the negative viewing direction. Since we transform the camera position instead of the ray entry position, this normal naturally varies. The camera position is transformed with M^{-1}. Thus, using the transformation rules for normals [23] the plane normal needs to be transformed with the transposed modelview matrix M^T. Initially, the plane normal is the negative viewing direction or $(0,0,1)^T$, the unit vector in positive Z-direction. That is, the sought-after vector is defined by the 3rd column of M^T. Again, reading columns is not supported; thus, we transpose the matrix again and instead read the respective row. Using this normal we then set up a virtual plane through the volume center and subsequently move it away from the camera along the negative normal direction to prevent intersections of the plane with the volume bounding box and then compute the ray/plane intersection in a straightforward way.

Knowledge of the ray/plane intersection does not suffice for determining the background pixel. Thus, the algorithm proceeds by computing the images $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ of the unit vectors in positive X- and Y-directions analogously to the way the plane normal is derived. Using these, in turn, dot products between the vector $\boldsymbol{d}$ from the volume center projected onto the plane to the intersection position and the two vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are computed (Figure 5.10). Writing down the dot

products yields $p_1 = |\mathbf{d}| \cdot |\boldsymbol{\nu}_1| \cdot \cos(\angle(\mathbf{d}, \boldsymbol{\nu}_1))$ and $p_2 = |\mathbf{d}| \cdot |\boldsymbol{\nu}_2| \cdot \cos(\angle(\mathbf{d}, \boldsymbol{\nu}_2))$. Since $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ are perpendicular, transforming the latter expression we further see that $p_2 = |\mathbf{d}| \cdot |\boldsymbol{\nu}_2| \cdot \sin(\pi/2 - \angle(\mathbf{d}, \boldsymbol{\nu}_2)) = |\mathbf{d}| \cdot |\boldsymbol{\nu}_2| \cdot \sin(\angle(\mathbf{d}, \boldsymbol{\nu}_1))$. That is, the two dot products return the abscissa and the ordinate of the ray/plane intersection with respect to the coordinate system spanned by $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$. Furthermore, the dot product just implements a projection of the difference vector $\mathbf{d}$ onto $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$, respectively. Thus, instead of the volume center projected onto the background plane we can equally well use any vertex on the line through the camera position and the volume center. In our implementation, the latter is used. These coordinates can then be directly used (or after applying some user-specified scaling factors) as texture coordinates for accessing a 2D texture storing the background image. Assuming the background image texture is bound to the third texture unit the final implementation of the background computation then is given by the following instructions:

```
# Compute the difference vector
SUB diffVec, intersection, center;

# Compute the texture coordinates
DP3 temp.r, diffVec, state.matrix.modelview.row[0];
DP3 temp.g, diffVec, state.matrix.modelview.row[1];
MUL temp.rg, temp, userScale;

# Look up the texel value
TEX temp.rgb, temp, texture[2], 2D;
MOV temp.a, 1.0;

# Blend the background pixel
SUB_SAT texblen, 1.0, dst.a;
MAD dst, temp, texblen.x, dst;
```

Volume Shader Examples

Figure 5.11 (a) shows an isosurface of the λ_2 scalar field already visualized in Section 5.3.3. Aside from being less prone to artifacts than a visualization with interpolated stripes, the visualization fails to illustrate the benefit of GPU-based ray-casting which is not just a qualitative improvement but also increased flexibility. Thus, e.g., once the isosurface has been hit by the sampling ray, the original isosurface implementation terminates the volume sampling process. But, of course, we can equally well perform the surface lighting computation but then re-orient the ray towards the light source to determine whether the isosurface is hit again, i.e. whether the surface point lies in shadow. Using a simple light-

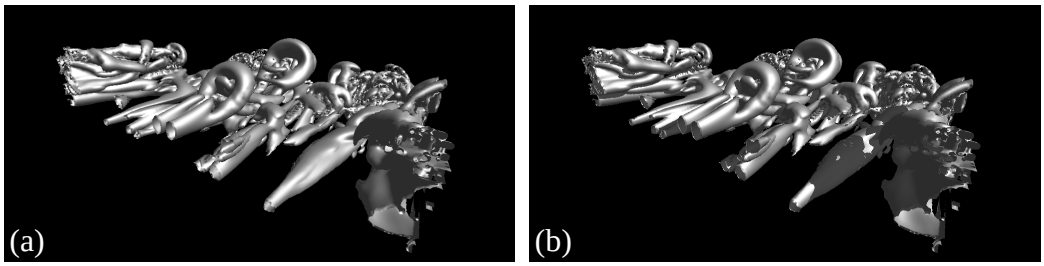

Figure 5.11 Isosurfaces of the λ_2 scalar field of K-type transition data visualized by GPU-based ray-casting. Adding self-shadowing (right) gives additional depth cues and aids in understanding complex geometries.

ing model discarding scattering effects we can then simply assign the pixel the ambient color to make it appear darker. Figure 5.11 (b) shows the final result. Obviously, shadows have considerable effect on the realism of the visualization and help in understanding complex geometrical structures.

Shadows can be integrated into slice-based volume rendering, too, albeit this requires significant effort [6]. In contrast, adding shadows using our flexible framework is straightforward. Similar arguments apply to various optimization techniques and other advanced volume visualization techniques.

Pre-integration (Section 2.4.1), e.g., usually requires two texture lookups, one for determining the scalar values at either end of the interval for which the contribution to the final image is to be determined. With our framework the numerical integration is coded explicitly which means we have full knowledge of the history along the sampling ray. Since the intervals are packed densely this means that the scalar value on the front face of a slab matches the scalar value on the back face of the neighboring slab and vice versa; thus, one component required for the lookup in the pre-integration is table is readily available from the previous sampling step.

Direct volume rendering also benefits from explicit programming in another way since early ray termination (that is, terminating the volume sampling once an opacity threshold is exceeded, the idea being that further sampling steps will only have negligible visual contributions to the final image) is trivially integrated by a set of two (due to the nested loop construct) conditional BRK instructions. In contrast, integrating this optimization into front-to-back slice-based volume rendering requires interrupting the slice blending at regular intervals to examine the alpha components of the pixels obtained so far by means of a fragment program and, where appropriate, setting the z-value such that the corresponding fragment is immediately discarded when resuming rendering [42].

For this work a series of enhanced visualization techniques was implemented exploiting this flexibility of single-pass GPU-based ray-casting. Since λ_2 isosurfaces, however, are synthetic and do not present any real physical object, none of

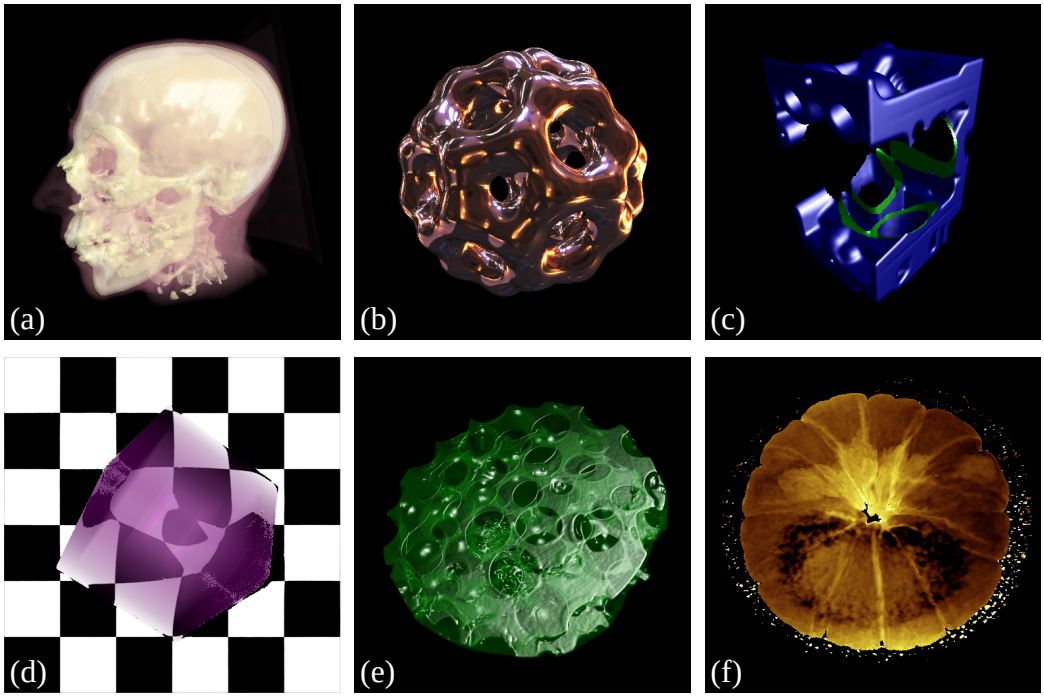

Figure 5.12 Various visualization techniques demonstrating the flexibility of GPU-based ray-casting. **(a)** Combination of isosurfaces and direct volume rendering; **(b)** image-based lighting; **(c)** volume clipping; **(d)** continuous refraction; **(e)** transparent isosurfaces; **(f)** translucency. Data sets courtesy UNC Chapel Hill, UC Davis, General Electric Corporation, University of Erlangen, and Lawrence Berkeley Laboratory.

these effects are actually useful for numerical flow visualization and rather apply to other areas of scientific visualization like medicine, engineering, and entertainment.

Regarding medicine an effective visualization technique is to combine isosurfaces and direct volume rendering to simultaneously visualize soft tissue and bone. For this, an isosurface is extracted as described in the previous section but, in addition, the volume rendering integral is numerically evaluated on the line segment from the ray entry position to the isosurface intersection. The contributions of the lit isosurface and the volume traversal are blended to yield the final pixel color (Figure 5.12 (a)).

Also relevant from a medical point of view but equally interesting for engineering applications is the possibility to visualize only selected parts of a data set (or, equivalently, to hide selected parts) to allow for examining the interior of objects. This is accomplished by deciding for each sampling position whether it lies

inside or outside a clipping geometry defined by the isosurface of another volume and subsequently performing the surface lighting and terminating the sampling or otherwise proceeding the search for isosurface intersections. Figure 5.12 (c) shows an engine block clipped against a radial distance field. An alternative approach to attain this goal is to use transparent isosurfaces. Figure 5.12 (e) illustrates this by the example of several individually lighted isosurfaces in a micro–computer-tomography scan of metallic foam.

Predominantly artistic or entertaining purposes serve visualization techniques modeling physical effects of the respective objects. Thus, if the ray is not reoriented towards the light source (as it is done for self-shadowing) but instead set to the reflection vector, we can use this vector for image-based lighting like *sphere mapping* [102]. As shown in Figure 5.12 (b) this can be efficiently used for modeling metallic or other highly reflective surfaces.

On the other hand, if we determine the distance light has to travel through the volume enclosed by an isosurface and use this distance to determine a color using a 1D texture, translucency effects can be modeled (again neglecting scattering effects). This effect is typically experienced when a semi-transparent object is lit from behind, thereby exhibiting bright silhouettes and dark areas where the object is dense or where the object thickness is high. Figure 5.12 (f) illustrates this at a volume data set of an orange.

Finally, when mapping scalar volume data to optical densities, the ratio of these densities at the current and previous sampling positions can be used in conjunction with the volume gradient defining a virtual surface to model continuous refraction by altering the ray direction. For this application just blending a textured quadrilateral for obtaining the background pixel is clearly insufficient; thus, this shader justifies the more elaborate background calculations of our approach. Figure 5.12 (d) shows a box data set in front of a checker board pattern visualized with our refraction shader.

Evaluation

The previous section illustrated that GPU-based ray-casting is superior to standard slice-based volume rendering with respect to flexibility. In order to give a performance comparison the implementation was compared to a standard slice-based volume renderer [18]. Due to the absence of volume rendering applications providing comparable functionality to what was shown above, we restrict ourselves to a comparison of the rendering performance for pre-integrated volume rendering. In general, standard slice-based volume rendering achieves framerates about 2–3 times higher than what is obtained with the given implementation. The primary reason for this performance loss can be identified to be the low performance of branching and looping instructions—the backbone of our implementation. Fur-

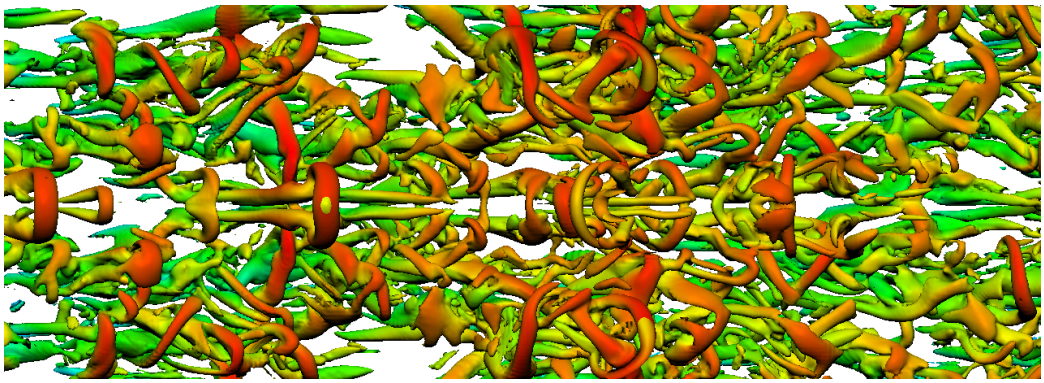

Figure 5.13 Isosurface of the λ_2 scalar field of K-type transition data colored with velocity magnitude. Although resulting in a data reduction, feature extraction can result in highly complex geometrical structures. Visualization generated with the commercial flow visualization software *PowerVIZ*.

thermore, early ray termination must be attested a very minor performance benefit, too, due to the overhead of the additional `BRK` instruction. The measurements, however, refer to an early stage of `ARB_fragment_program2` and, thus, must be considered preliminary. In addition, since texture-based volume rendering runs at highly interactive framerates, a framerate loss of 50 percent is bearable, meaning the system is still interactive for most practical applications—for viewport sizes of 512×512 pixels standard pre-integrated volume renderings of the data sets shown in the upper row of Figure 5.12 are generated at 7.2 fps, 14.0 fps, and 10.8 fps, respectively, on an NVIDIA GeForce 6800 GT graphics accelerator card. Thus, in environments where quality and functionality are rated higher than pure rendering performance the framework still presents an alternative well worth being considered.

5.5 Vortex Segmentation

In the exposition given so far, the λ_2 algorithm was implemented both in software and—for attaining interactivity—also by exploiting graphics hardware. Furthermore, visualization techniques were examined allowing for high-quality visualizations of the resulting λ_2 scalar field. The λ_2 algorithm, however, was employed as-is. On the one hand, this is well justified because as seen in Section 2.3.2 the λ_2 criterion in its original form is a local vortex detection method classifying a grid point according to its affiliation to a vortex independently of its neighbors—without this property an efficient GPU implementation as described in Section 5.3 would not have been possible. On the other hand, although feature extraction in

general leads to a data reduction, this data reduction is limited. Thus, with increasing data set sizes but static feature extraction techniques and human cognition capabilities we will observe diverging tendencies finally resulting in a value loss of high-resolution flow data. Figure 5.13 illustrates this problem: the data set comprises only about seven million voxels; analyzing the data, however, proves highly difficult in practice due to an overwhelming number of interwoven vortices. Thus, what helped for the GPU-implementation now presents a major hindrance for efficient data analysis since no list of individual vortices is generated which forces the researcher to analyze the data in its entirety. That is, using the λ_2 method as-is also has drawbacks from a practitioner's point of view.

Global vortex detection methods extracting core lines like the predictor-corrector vortex detection approach introduced in Section 2.3.2, in contrast, naturally do not suffer from this problem but are considered inferior to the λ_2 method for the shown transitional flow by our collaborators. A vortex detection method combining accurate vortex detection in the manner of the λ_2 criterion and auto-segmenting capabilities inherent to global vortex detection methods is thus highly welcome.

In this section we discuss a hybrid approach of local and global vortex detection methods combining the λ_2 method with the predictor-corrector approach leading to a system matching these criteria and yielding a more useful representation of the detected vortex structures. The resulting system allows for dissecting the flow by segmenting vortices and, thus, enables the researcher to concentrate on the most interesting flow regions. The work was first published with U. Rist [96].

5.5.1 Related Research

The need for improved analysis tools has been observed before. In [21] and [87] systems are presented for identifying, classifying, visualizing, and automatically tracking observable flow field features. These works concentrate on the understanding issue and identify vortices based on a region-growing approach on the vorticity magnitude field (Section 2.3.2). However, the effectiveness of the former depends on the sophistication of the latter. That is, to be useful in practice more advanced vortex detection methods must be employed.

Vortex segmentation based on the λ_2 criterion was first presented in [74]. Again, a region-growing algorithm is used for object identification. Thus, although the feature recognition rate is improved as compared to a vorticity-based approach, no optimal results are obtained since domain knowledge is not taken into account. From an algorithmic point of view the segmentation will, therefore, yield identical results independent of whether the scalar field has been derived from MRI images or the λ_2 method. In addition, no interaction functionality is included in the system, limiting its practical use.

Also based on the λ_2 criterion is the Galilean-invariant vortex core line extraction proposed in [81]. This algorithm, which was developed in parallel to and independently of the vortex segmentation approach described in the following, extracts vortex cores by tracing ridge or valley lines of the λ_2 scalar field. A major drawback of this method is the requirement for 2nd order derivatives. Furthermore, while several visualization techniques of the resulting vortex core lines are explored, the paper lacks a discussion of the interaction and data analysis part.

The system devised for this work addresses both the feature detection aspect by separating the λ_2 vortex structures in consideration of domain knowledge and the user interface part. The latter component has been developed in cooperation with practitioners to provide the functionality required in everyday work while, on the other hand, keeping the system as lean as possible to allow for fast and easy use.

5.5.2 Adaptation and Implementation

Obviously, since searching for low-pressure regions in the predictor-corrector algorithm serves the only purpose of a vortex core indicator, *any* scalar field signifying a vortex will equally well do. And so does the λ_2 scalar field. In addition, pressure is a scalar property often unavailable—especially for experimentally obtained data—while λ_2 values can be calculated from the most elementary flow field information: the velocity field. And, as argued in Section 2.3.2, low pressure is not even a reliable indicator for vortices either. Thus, the predictor-corrector approach complements the λ_2 method in an ideal way.

Our algorithm starts with an initialization phase classifying grid points according to their λ_2 values to build a set of seed points. Since λ_2 values become more negative the closer the core, only grid points with negative values defining local minima qualify as seed points. Whether a grid point will actually serve as starting point for a vortex core is unclear since it might be eliminated if enclosed by another vortex tube. Thus, the set resulting from the initialization phase includes only *potential* seed points. In practice, the fraction of potential seed points obtained this way is very small—for the data set of Figure 5.11 only about 4 percent of the original grid points.

The actual process of growing the vortex core is then similar to the original algorithm presented in [5], i.e. a seed point is picked, λ_2 is minimized on the plane through the seed point perpendicular to vorticity, and, finally, the vortex skeleton is grown in both directions (Figure 5.14). When choosing a seed point we always select the one with the smallest (most negative) λ_2 value. Since a more negative λ_2 value indicates stronger swirling flow, this will automatically yield a list of vortices sorted by strength.

For each predicted position we first determine whether the position lies inside

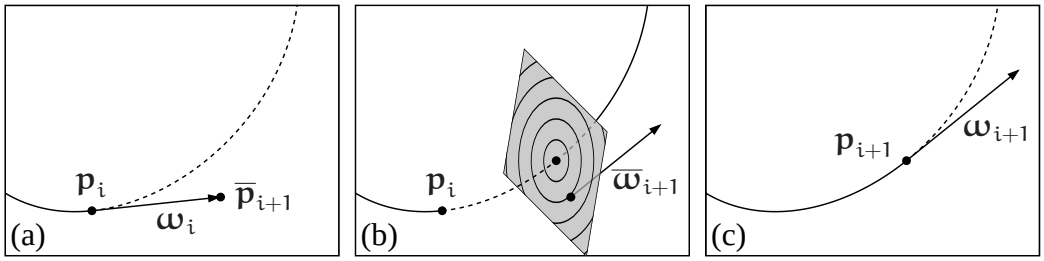

Figure 5.14 Vortex core detection using the combined predictor-corrector/λ_2 approach. Starting from a point $\mathbf{p}_i$ on the vortex core the vorticity $\boldsymbol{\omega}_i$ is first determined for predicting a new position $\overline{\mathbf{p}}_{i+1}$ **(a)** which is then corrected by minimizing λ_2 in the plane through $\overline{\mathbf{p}}_{i+1}$ and perpendicular to $\overline{\boldsymbol{\omega}}_{i+1}$ **(b)** to obtain $\mathbf{p}_{i+1}$ from where the next iteration is started using $\boldsymbol{\omega}_{i+1}$ as core direction indicator **(c)**. Figure inspired by [5].

any vortex tube (including the currently built) in which case the core is terminated and added to the list of detected vortices. If the trace needs to continue we minimize the λ_2 value in the plane defined by vorticity at the predicted position to correct the guess. During this growth process it is very unlikely that a position—whether predicted or corrected—matches a grid point; thus, interpolation is required. In particular, if we do not want the local minima to be confined to grid points some higher-order interpolation is required. Therefore, a four-point Lagrange interpolation as introduced in Section 2.2.2 is utilized (as it is used in the original implementation). While this interpolation is of high quality, it is also computationally expensive which presents a major drawback for the minimization problem since many optimization algorithms are based on the gradient whose determination, in turn, requires several function evaluations. In our application this translates to prohibitively many Lagrange interpolations and renders traditional optimization techniques inappropriate. We, therefore, employ a direct search approach [32] which is a brute-force approach not resorting to any analytical aids like gradients. Instead, the algorithm inspects various neighborhood samples for guessing a promising direction and then reuses the result to find the most promising direction more quickly in the next iteration. As the method only requires most basic operations (function evaluations) it has been shown to be robust and to also work for problems where steepest-decent methods fail.

Once a core vertex has been detected we radially sample the plane perpendicular to the vortex core until a given λ_2 threshold is exceeded and encode the cross section by means of Fourier coefficients as described in Section 2.3.2. Figure 5.15 (b) shows the result of the vortex separation when applying the proposed hybrid vortex detection to the transition data set already used in the previous sections. For the given images, 10 rays where used for sampling the cross

section and $n = 5$ coefficients of the Fourier expansion where saved. For displaying the vortex geometry, the function representing the radii table is evaluated for 16 equidistantly-spaced angles. Hue is used to differentiate between the vortices—the weaker a vortex (smaller absolute λ_2 value), the greater the hue in HSV color space.

Comparing the separated vortices (the extraction takes about 68 s on an AMD Athlon 64 processor 3500+) with a standard λ_2 isosurface extracted for the very isovalue used for determining the cross sections (Figure 5.15 (a)) reveals a very close match of the two visualizations (Figure 5.15 (c)). In fact, there are only two minor differences. First, some structures near the volume boundary are missing in the separated visualization and, second, some additional fine structures evolve from the vortex tails.

The structures referred to in the first case are caused by inaccurate one-sided derivatives at the volume boundary (Section 2.2.1). These structures should not be classified as vortices. And they are not—as is seen in the image—so errors made during the λ_2 computation do not find their way into the final visualization.

In the second case, the additional fine structures are vortex core lines that have been obtained by allowing for skeleton growth also beyond vortex tails. This is another benefit when applying the predictor-corrector vortex detection method to a λ_2 scalar field since it enables the researcher to identify connected structures hitherto visualized as separate vortices.

To further verify the quality of the presented approach, Figure 5.16 shows a particle trace computed for the single Λ-vortex already used for the comparison of the CPU and GPU implementations of the λ_2 method (Section 5.4). For the image, the particle probe was placed near the detected vortex core and a trace computed by integrating the velocity field utilizing RK45 integration (Section 2.3.2). Ideally, the resulting trace should exactly enclose and follow the core line detected during the vortex segmentation process. In the example, this requirement is essentially fulfilled and inaccuracies are only seen on a short core segment where the particle travels parallel to the vortex core (see close-up). The particle leaves the core through the appendix shown in the upper right corner near the Ω-shaped vortex segment. This is correct behavior and must be ascribed to the strong velocity component along the longitudinal axis. As for the interpolations involved in tracing the core line and determining the vortex tube cross sections, a four-point Lagrange interpolation (in contrast to a less costly trilinear interpolation) is required in order to obtain accurate results.

As argued in Section 2.3.2 the eigenvector corresponding to the real eigenvalue of the velocity gradient tensor is a more reliable indicator for estimating core directions; thus, tracing the core line based on the eigenvector might be assumed to result in a closer match of the particle trace and the extracted core line than shown in Figure 5.16. The eigenvector, however, is not uniquely defined with

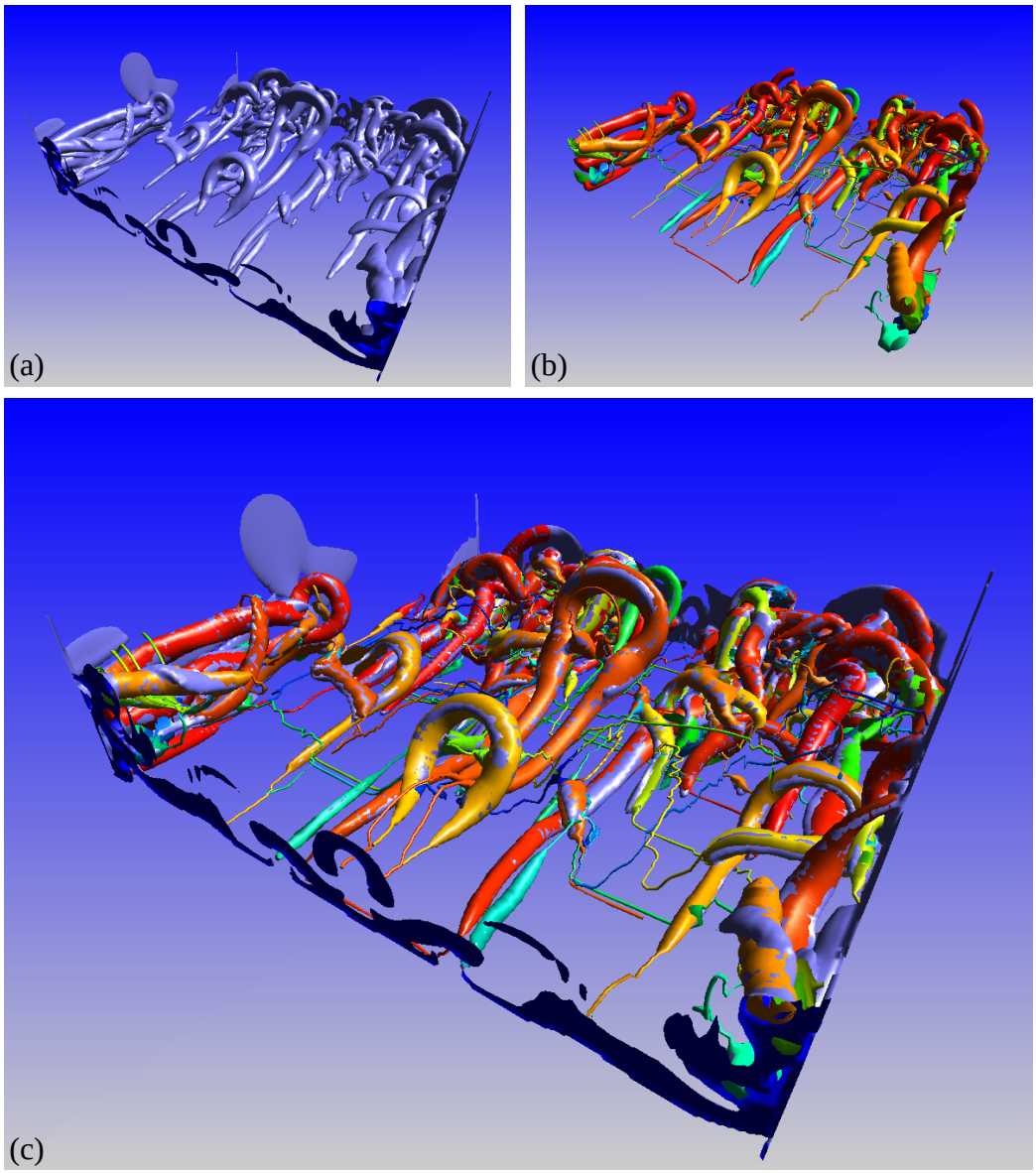

Figure 5.15 Comparison of λ_2 isosurface and vortex structures detected and separated by combining the λ_2 criterion with a predictor-corrector vortex detection method. **(a)** λ_2 isosurface, **(b)** vortices separated with the hybrid approach. Image **(c)** shows both the isosurface and the separated vortices.

respect to sign and thus cannot be used as-is for following the vortex core. We propose a simple solution to this problem in that we re-orient the eigenvector based on the sign of the dot product of the eigenvector and the vorticity vector. Our comparisons, however, show that the visualizations hardly differ from what is

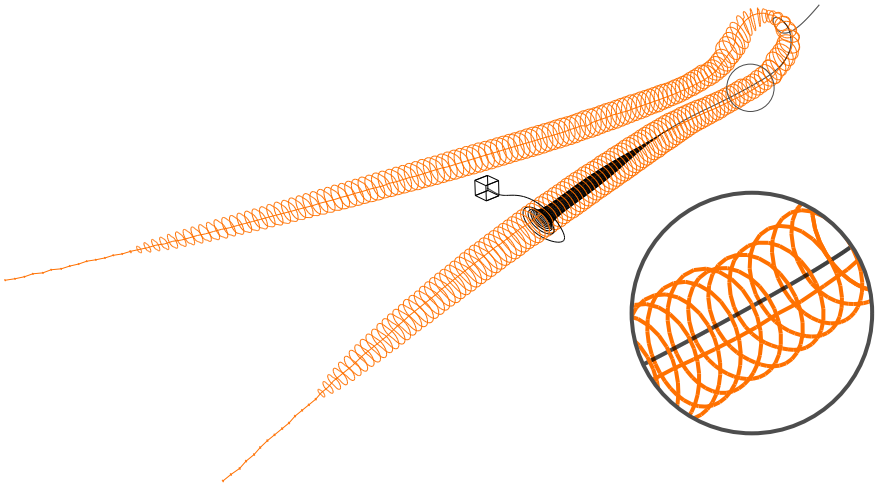

Figure 5.16 Particle trace computed on the velocity field. While in general the trace encloses the extracted vortex core, in some regions the trace is parallel to the core line (see close-up).

obtained with a vorticity-based estimator and that there is only a small discrepancy in the number of detected small-scale vortices of about 7 percent for the data set of Figure 5.15. Responsible for this low discrepancy is the corrector step which returns to the exact core line fairly independent of the quality of the prediction. In consideration of the high complexity involved in the computation of eigenvectors, using vorticity thus seems a valid alternative. A detailed comparison can be found in [96].

5.5.3 The Vortex Browser

The vortex set shown in Figure 5.15 comprises more than 300 vortices. This includes weak and strong vortices, vortices with short and long cores, and vortices which cover small and large regions of the simulation volume. Obviously, some filtering must be applied before the data can be thoroughly and efficiently analyzed. Typical questions arising during the analysis are: What is the volume of a certain structure? What is its swirling strength, circulation, and vorticity? How about induced velocity, vortex stretching, and dissipation? None of these questions can be answered when considering the vortex set in its entirety; thus, means are required to manipulate the vortex for which purpose we provide a *vortex browser*. For a practical tool, however, manipulation functionality must be coupled with various numerical flow visualization techniques.

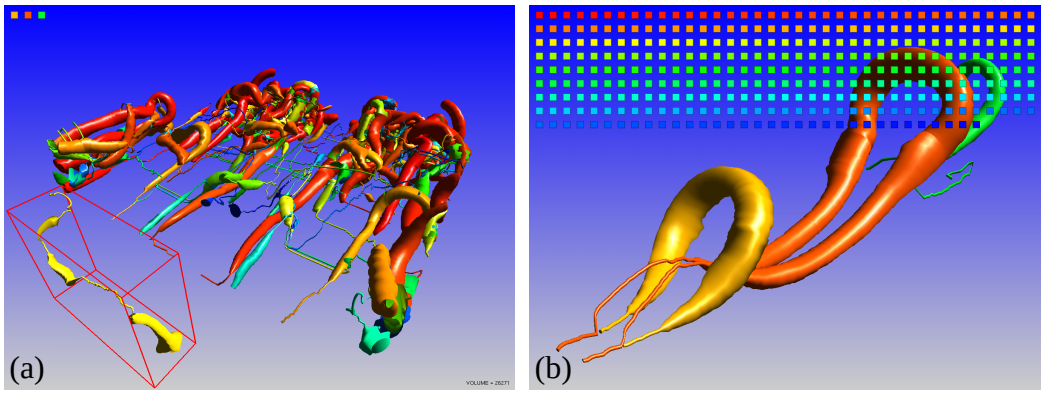

Figure 5.17 Manipulating the vortex set. **(a)** Hiding the three central Λ-vortices and selecting a single vortex for closer examination. **(b)** Inverting the selection. Using the colored buttons individual hidden vortices can be temporarily or permanently re-inserted.

Interaction Techniques

To be able to interact with vortex structures we insert the individual vortices into a custom scenegraph, i.e. a graph structure of transformation and objects nodes that is traversed for rendering, which allows for simple manipulations like picking and transformations. This, however, is insufficient since researchers either want to see a certain vortex or they do *not* want to see certain vortices being irrelevant for the special application or obscuring the region of interest. Our system, therefore, provides hiding and inversion functionality (Figure 5.17). In image (a) the central Λ-vortices have been hidden. Another vortex—one that would have been visualized as two separate structures with standard λ_2 isosurfaces—has been dragged out of the flow and is shown (with a selection box) in the front. Its volume is shown in the lower right corner. For image (b) the selection has been inverted in order to allow for a closer examination of exactly the hidden vortices. In either case, for each hidden vortex a button colored with the respective vortex color is inserted. By moving the mouse cursor over any of these buttons, the corresponding vortex is shown at its original location. By clicking the buttons, the corresponding hidden vortex is permanently re-inserted into the visualized vortex set. Of course, the buttons can be disabled if desired.

Manually defining the vortex set may become tedious if there are dozens or even several hundreds of vortices. However, since the flow dynamics is dominated by the largest vortices, this task can be partially automated. In the system, this functionality is provided by means of a length filter which can be used to eliminate smaller and weaker vortices.

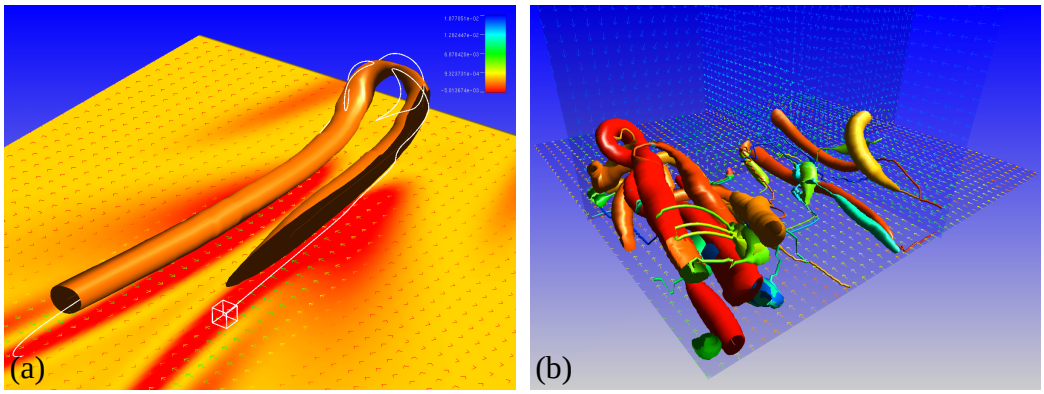

Figure 5.18 **(a)** Vorticity line traced from near the vortex core. The necessity for a correction step in tracing the core is clearly seen at the Ω-shaped rear part. **(b)** Clipping and vector plot (hedgehog) visualization.

Visualization Techniques

From an information visualization point of view the capability to extract individual vortices defines the focus. The vortices, however, have been extracted from a flow field and this context should not be lost. To provide context information, too, we incorporated a number of standard numerical flow visualization techniques described in Section 2.3.2. Particle tracing by means of a freely positionable probe was already illustrated in Figure 5.16. As discussed in Section 2.3.2, particle traces are usually computed by integrating velocity. This, however, can be dangerous and result in misleading visualizations since velocity is a function of time which means that taking a snapshot of the flow for computing the trace is simply wrong—no real particle will ever follow the presented stream line. Therefore, the particle should only be used in regions in which the flow is known to exhibit low temporal rates of change. For the given isolated Λ-vortex, this condition is met near the "legs". To account for these problems, we also integrated *vorticity lines* into our system which are computed analogously to particle traces by integrating the vorticity field (Figure 5.18 (a)). In contrast to particle traces, vorticity lines are meaningful in snapshots of the flow field, too, since they follow the vortex core when integrating with an infinitely small step size (or, as seen above, when using a finite step size in combination with a corrector step). However, since again shear layers are interpreted as vortical regions, vorticity lines alone are no useful tool. In practice, a combination of field lines of the instantaneous velocity field *and* the instantaneous vorticity field, is preferred. For the study of time-dependent flows (which the presented vortex browser does *not* support) replacing stream lines by streak lines would be an obvious improvement.

In either case, as field lines reveal only meaningful information when posi-

tioned carefully—which, in particular, also applies to vorticity lines [54]—a more intuitive technique is needed to visualize the velocity and vorticity fields. For this purpose, movable slices with vector plots of variable resolutions have been integrated into the system. Although this visualization is simple and dated, it is nevertheless the predominant numerical flow visualization technique for visualizing vector fields obtained by physical measurements. Figure 5.18 (b) shows a screenshot of the visualization technique applied to K-type transition data. As seen in the image, slices can also be used to clip unwanted parts of the visualization.

When vector plots are insufficient, dense representations of the original vector field are often more useful. Therefore, line-integral–convolution (LIC, Section 2.3.2) visualizations of the flow projected onto any movable slice can be shown. For a snapshot of the flow, however, LIC again relies on the computation of streamlines and thus also suffers from the problems already mentioned above, so care should be taken. An example visualization using LIC in combination with a dense representation of an associated scalar field (shear stress) is given in Figure 5.19 (a). To allow for a quantitative analysis a legend is provided in the upper right corner.

Shear stress is of special importance for the understanding of vortices since, as seen in Figure 2.8, Section 2.3.2, shear layers give birth to new vortices. We have, therefore, integrated into the system standard Marching Cubes isosurface extraction as described in Section 2.4.2 which allows for mixing the visualization of separated vortices with an isosurface visualization of this important phenomenon. An example is shown in Figure 5.19 (b). The required shear layer computations are based on the second invariant I_2 of the strain-rate tensor S [52]:

$$\begin{aligned} I_2 &= \frac{1}{2}\left(S_{ii}S_{jj} - S_{ij}S_{ij}\right) \\ &= S_{11}S_{22} + S_{22}S_{33} + S_{11}S_{33} - S_{12}^2 - S_{13}^2 - S_{23}^2 \,. \end{aligned}$$

Since the strain-rate tensor is required for computing the λ_2 scalar field, anyway, computing the shear layer is virtually a by-product which comes almost for free.

Finally, we reused the frustum calculations derived in Sections 2.7.1 and 3.5 to integrate red/cyan anaglyph stereo to ease analyzing complex interwoven structures which may result from certain isovalues. Figure 5.20 gives a screenshot. Although it would have been possible to reuse not only the frustum calculations but the complete generic stereo library developed in Section 3.5, this alternative was discarded since we have full control of the program and because systems comprising only a single component are generally easier to handle and more accepted.

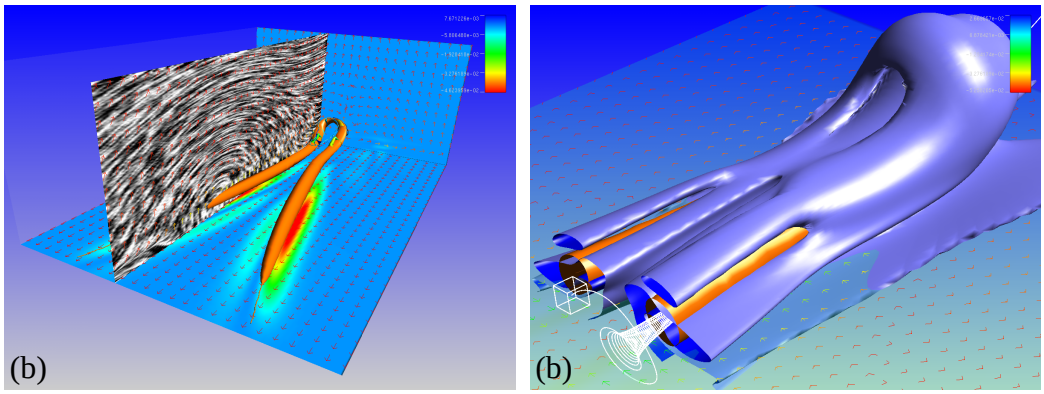

Figure 5.19 **(a)** Dense vector field representation with LIC and color encoding of scalar field (shear stress), **(b)** Combined visualization of shear layers, cutting planes, arrow plots, and particle traces.

5.5.4 Evaluation

The vortex browser was developed in close cooperation with fluid dynamics engineers and has been evaluated by this group of experts and practitioners. Both benefits and problems of the system were found.

The main benefit turns out to be the novel combination of reliable vortex detection (exposing also connected components) and selective visualization, a combination allowing for a great complexity reduction necessary for analyzing current simulation data. Once vortices have been extracted, the structures can be picked, hidden, extracted, and freely moved and rotated. Of course, the latter is standard functionality also found in more general visualization packages used so far (Tecplot, COVISE) and, indeed, the vortex detection part can be decoupled from the system to provide a set of vortices to be visualized with standard tools. However, in contrast to standard tools our system is problem-oriented; thus, analyzing flows with our tool has been reported to be significantly more efficient than with general tools from a usability point of view (intuitive, adapted user interface) and a technical point of view (speed)—both are important aspects of numerical flow visualization.

The major drawback of the given hybrid approach is that the high performance of local vortex detection algorithms is inevitably lost. Thus, while the GPU-based λ_2 implementation of Section 5.3 is highly interactive, separating vortices is a time-consuming process that may even require minutes for data sets comprising hundreds of vortices. Practical experience, however, shows that this temporal investment is greatly compensated for by the time subsequently saved during the data analysis.

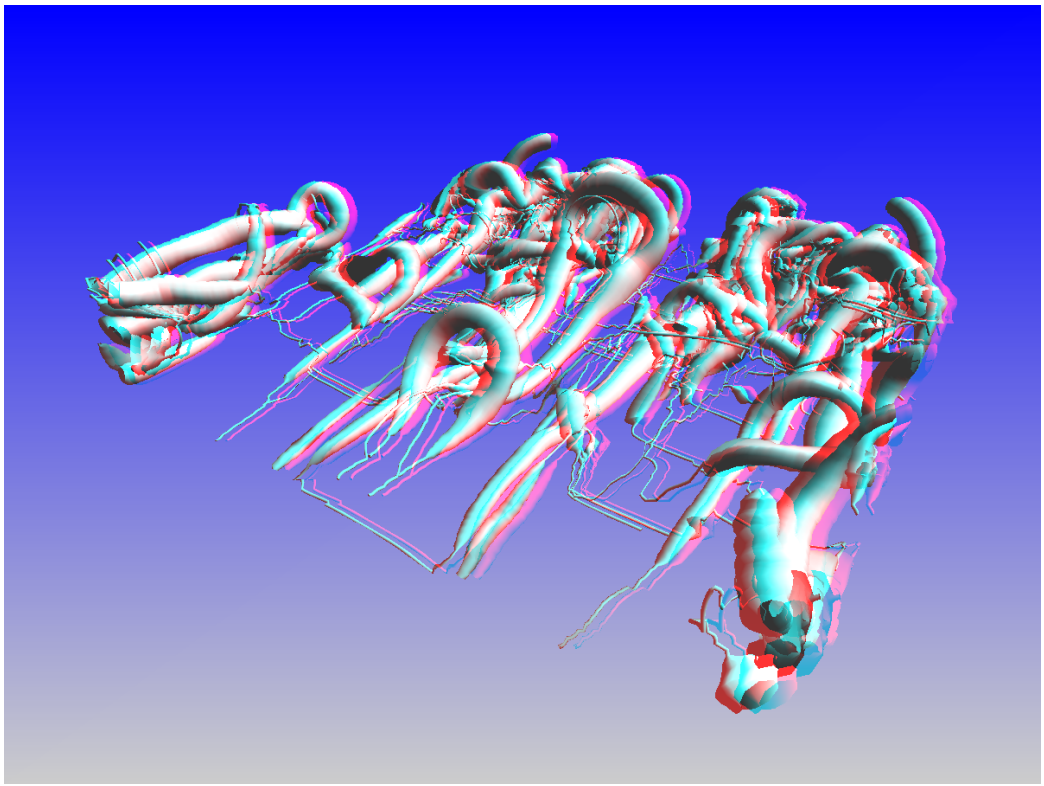

Figure 5.20 Red/cyan anaglyph visualization of separated vortices.

Chapter 6

Conclusion

In this work, a unique multi-level approach for accelerating numerical flow visualization has been presented. Looking back, the work reflects the developments in computing hardware over the last years.

Initially, only dedicated, highly expensive graphics computers provided sufficient memory, computing power and graphics resources for interactive scientific visualization—and numerical flow visualization in particular. These machines had to be shared among researchers which gave birth to remote visualization solutions. The basic idea was, on the one hand, to use local hardware exclusively for displaying the final visualization and for communicating with the user while, on the other hand, to exploit graphics and computing resources of distant computers connected to the local machine by some network for executing the visualization application and for storing data sets. Remote visualization solutions already existed before this work was begun; in contrast to any existing approaches, however, the solution developed for this work was the first truly generic approach requiring no modifications of the visualization applications at all in order to make them network-aware. Since the available network bandwidths and client and server resources with respect to both GPU and CPU power and screen space in practice will be highly diverse, several strategies were devised for rendering existing applications usable in any of these environments. This includes providing generic stereo visualization and user interface adaptation solutions and alternative system architectures for maximizing framerates. Using the proposed remote visualization system, researchers are neither forced to visualize different data nor to employ different visualization techniques or visualization applications; thus, the solutions proposed in Chapter 3 present the least intrusive level of acceleration techniques for numerical flow visualization of this work.

When main memory and disk storage capacities increased and—driven by the computer games industry—both computing and processing power of desktop computers improved, local hardware started to become an attractive visualization

platform, a welcome alternative to remote visualization in environments providing only high-latency, low-bandwidth networks. Depending on the data set structure, however, no interactive work was possible with real-world data sets found in the car manufacturing industry—and this still applies to unstructured grids using even current hardware. This motivated the work presented in Chapter 4. The basic resampling approach described therein is well known from scientific visualization as a probing technique commonly used for data reduction and for opening ways to specific visualization algorithms. In this work, however, it was demonstrated that tailoring a resampling application to actual needs of engineers poses a major difficulty demanding adaptive algorithms, careful memory management, and parallel processing. Resampling presents itself as a mediator between non-intrusive acceleration techniques like remote visualization and highly-intrusive acceleration techniques changing the data attribute types and, accordingly, the visualization technique.

Two recent technological advancements are also reflected in this work. First, the advent of cluster computers and, second, the introduction of programmable graphics hardware. The ubiquitous availability of cluster computers comprised of cheap off-the-shelf hardware has decoupled the growth of processing power of single CPUs from the growth of computing time available for scientific simulations. The resulting imbalance between data set sizes and visualization capabilities motivated the demand for feature-based numerical flow visualization elaborated on in Chapter 5, research concentrating on the extraction of characteristic flow features to give a highly condensed representation of the relevant information contained in the underlying vector field. For this purpose, existing vortex detection algorithms were employed. This work, however, contributes several significant improvements to these algorithms. For one thing, reliable vortex segmentation was combined with state-of-the-art vortex detection—novel functionality eagerly awaited by fluid dynamics researchers. And by exploiting programmable graphics hardware, the performance of both vortex detection and visualization was improved to provide the first interactive vortex detection system for noisy flow data. Taking advantage of any of the presented feature extraction functionality requires abandoning both existing visualization applications and traditional flow visualization techniques. Feature-based flow visualization thus presents the most intrusive level of acceleration techniques.

It is seen that some levels of acceleration accelerate the visualization itself while others aim at accelerating the data analysis, the final goal of visualizing flows, as well. A number of techniques discussed—in general all generic approaches but also the work on volume visualization—can be used in contexts other than flow visualization, too. Many solutions proposed allow for interactive work, certain algorithms, however, are too computationally expensive (resampling, vortex segmentation) and should be used for pre-processing only. We shall give some

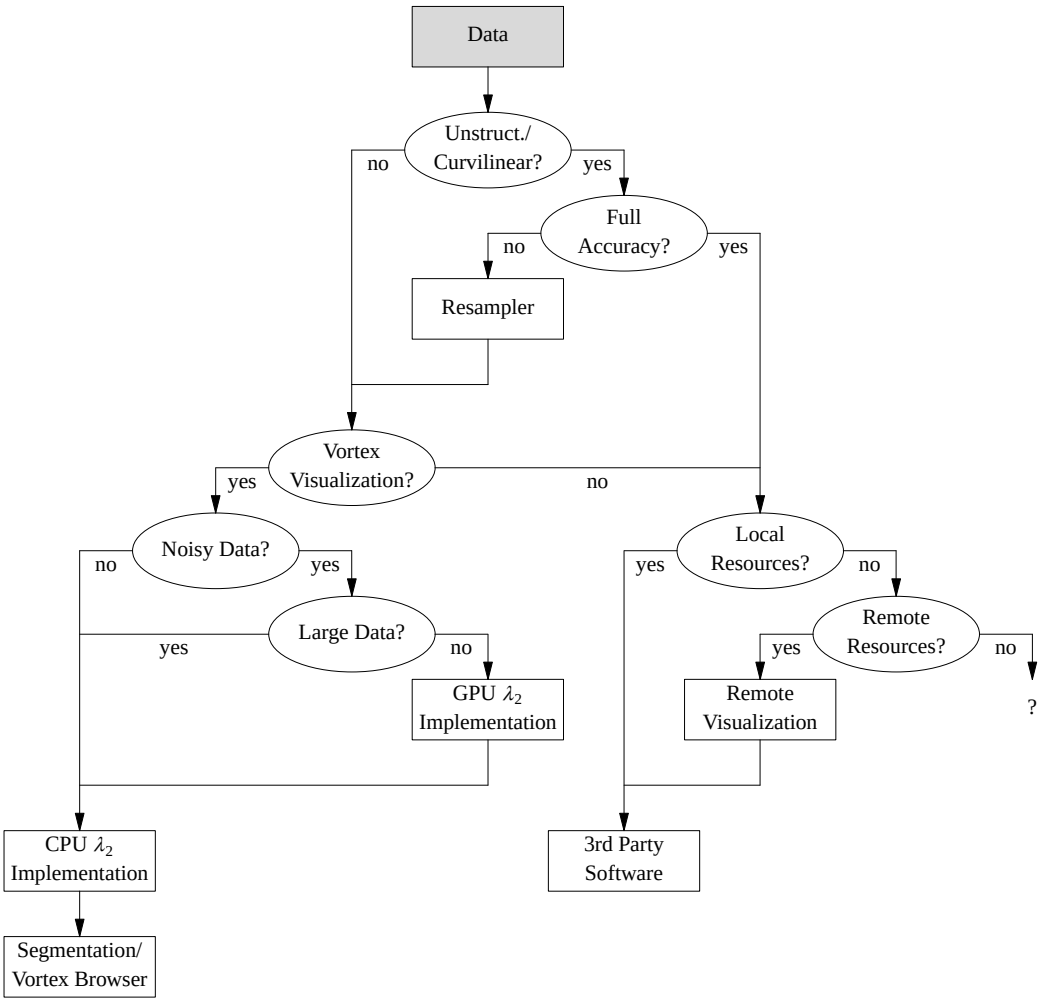

Figure 6.1 Flow diagram giving recommendations on how to employ the techniques presented in this work.

final recommendations on how to employ the outcome of this work and to aid in the decision in selecting a technique for the respective case. We will give the recommendations in form of a flow diagram yielding a recommended technique or work flow based on a number of decisions accounting for the data set size, the underlying grid type, accuracy requirements, visualization goals, and hardware resources available at both the local workspace and at remote locations (Figure 6.1).

We start by assuming a certain data set is to be visualized. As in this work no efforts have been made to directly visualize non-regular grids the question to be answered first is whether the data is defined on a curvilinear or unstructured grid.

If so, we need to bring to mind any accuracy requirements applying. When full accuracy is required, we must not modify the grid type and have to resort to any 3rd party software capable of visualizing the respective grid. Whenever possible, we would recommend to use local hardware resources for this purpose since this results in the smallest possible latencies.

In contrast, if resources available locally are insufficient—be it CPU power, the amount of main memory or disk space, or graphics capabilities—but if, on the other hand, appropriate remote hardware is available and accessible, we recommend to employ any of the remote visualization solutions presented in Chapter 3. As discussed, remote visualization is also advantageous in environments where multiple researchers have to access the same data since data replication is avoided. Therefore, we recommend remote visualization for this scenario, too.

In the worst case, full accuracy is required but neither local nor remote hardware is capable of visualizing the data in its original format. None of the proposed solutions is applicable to this scenario. We leave it as an open question what is to be done in this situation.

The most benefit can be gained from dropping the requirement for highest accuracy. We, therefore, recommend to resample the original grid to a hierarchy of regular grids or a single regular grid to clear the way for more efficient algorithms and GPU-supported implementations. This is independent of whether vortices are to be visualized or whether the velocity field is to be visualized using classical flow visualization techniques—the question to be answered next.

If we want to visualize the original data, i.e. the velocity field, the classical flow visualization techniques integrated into the vortex browser described in Section 5.5.3 can be used. However, since this tool includes only the most basic visualization techniques, we rather recommend to employ again some 3rd party software, especially PowerVIZ, which was shown to provide highly efficient algorithms for hierarchies of regular grids. Again, remote visualization can be taken advantage of in case the local resources fail to provide interactive framerates. The question arises what is gained in this case—after all, we returned to the very branch “Local resources sufficient?” of the decision graph already entered above. The answer is that by the resampling step, the probability that the answer will be in the negative again is reduced. And, accordingly, the probability of being able to attain favorable low-latency visualizations exploiting local hardware is increased.

In contrast, if features are to be visualized—which, judging from our experience, is inevitable for large data sets—the proposed vortex detection and segmentation algorithms should be utilized. For noisy data of comparably small size (i.e. data fitting into texture memory, translating to about 16 million voxels for the emerging generation of graphics cards with 512 MB on-board memory) we recommend the proposed GPU-based implementation of the λ_2 criterion (Section 5.3) as this solution allows for unprecedented performance in handling the

cycle of filtering, vortex detection, and visualization.

If texture memory is unable to accommodate the data, the cycle phases for determining optimal filter characteristics must be accomplished in software. In either case, we recommend using the proposed hybrid vortex detection approach (Section 5.5) for the final visualization since it allows for the most effective data analysis. Thus, in this case the GPU-based implementation is degraded to a helper application for determining suitable filters and isovalue settings.

Practical experience shows that the given work flow has opened the door towards more efficient and effective numerical flow visualization. Due to the chosen multi-level approach, the researcher can freely choose the degree of interference. It can be none at all, i.e. the researcher gets exactly the same visualizations as before (remote visualization), he can choose less accurate results in favor of increased performance (speed vs. quality trade-off with resampling), or he can speed-up the visualization by visualizing highly condensed information by means of flow features. Since the performance of graphics hardware grows even faster than what Moore's Law predicts for the performance of CPUs, utilizing GPUs—as done for accelerated feature extraction and advanced volume visualization—seems to be a future-proof solution. However, this only applies to processing performance and not to memory sizes. We have, therefore, made first experiments regarding the extraction of flow features from vector fields compactly represented by radial basis functions [104]. The results of these efforts, however, are currently not very encouraging for two reasons. First, we found that the encoding is not yet satisfying; thus, the reconstructed vector field suffers from accuracy problems. And second, although the vector field reconstruction is accomplished on a per-fragment basis utilizing the GPU, it is nevertheless slower than expected. Consequently, albeit being highly interesting from a conceptual point of view, we currently see no benefit for flow visualization resulting from this approach.

In the end, the rapid growth of data set sizes thus only seems controllable if the presented flow-visualization–related solutions, too, are adapted for distributed processing on GPU clusters as it has already been done for scientific volume visualization. Growing demands for visualization solutions for time-dependent data multiplying data set sizes by another three orders of magnitude support this prediction.

Color Plates

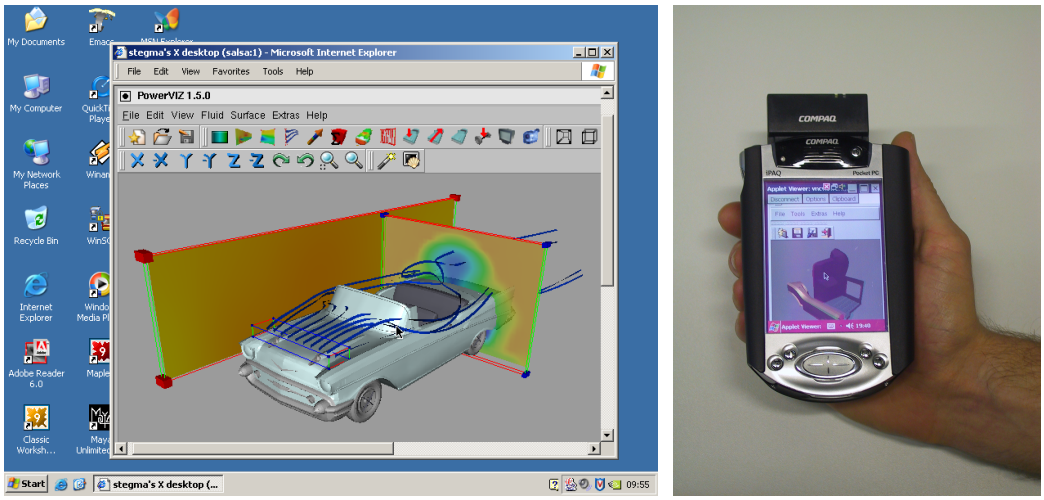

Figure 3.3 Remote visualization applications. Left: Flow visualization software operated in a web browser. Right: Post-processor for crash simulations operated on handheld computer. A Linux-based render server is used in both cases.

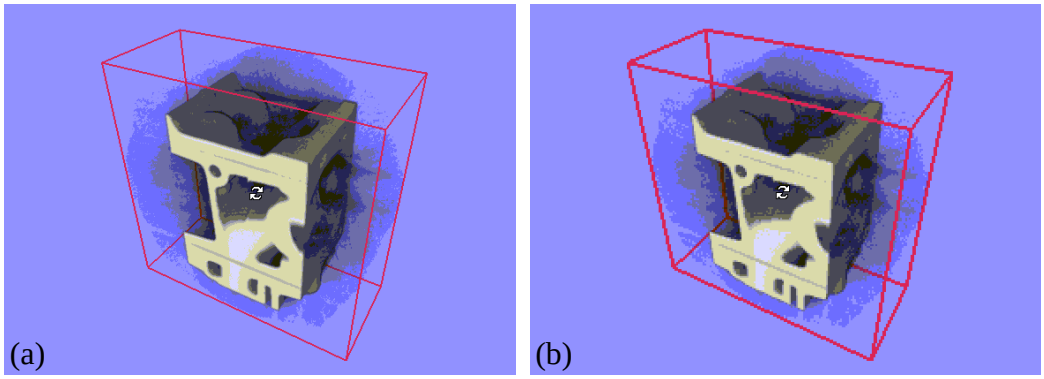

Figure 3.5 Motion handling by the example of a volume renderer. Downsampling effects are clearly visible at the bounding box but almost invisible in the volume. Note that the mouse cursor in the center is shown in high resolution in both images since it is not part of the rendered image.

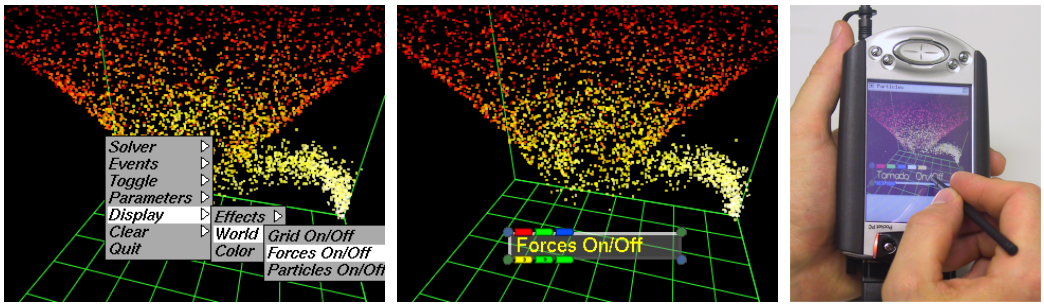

Figure 3.10 Comparison of original GLUT menu and adapted user interface. The application running on a Compaq iPAQ handheld PC is shown on the right.

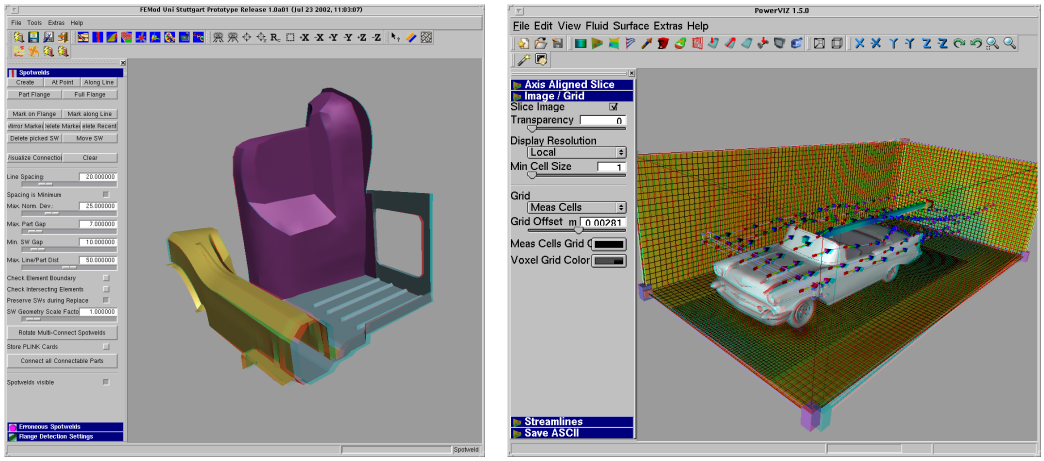

Figure 3.12 Stereo visualization using the generic stereo library. Left: *FEMod*, a commercial pre-processor for crash simulations; right: the commercial flow visualization software *PowerVIZ*.

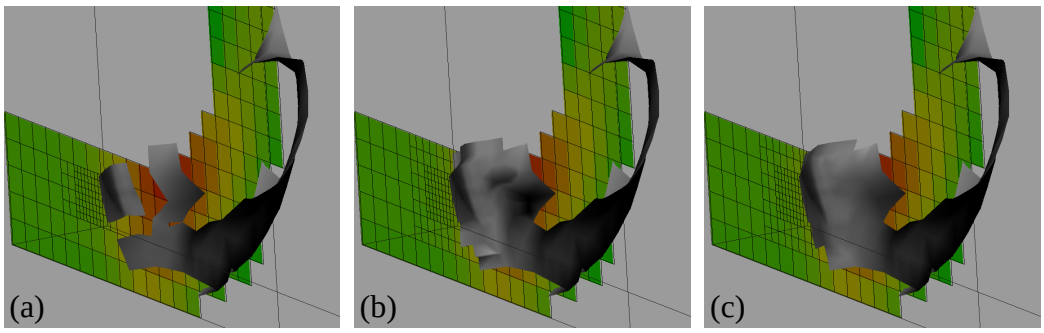

Figure 4.6 Comparison of isosurfaces calculated by PowerVIZ. **(a)** Uncorrected octree, **(b)** corrected octree without interpolation (subdivision only), **(c)** full correction with interpolation.

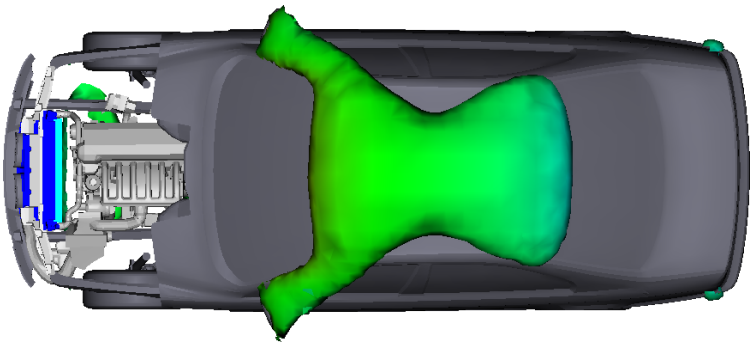

Figure 4.8 Isosurface computed with PowerVIZ using the adaptively resampled grid. The scalar property is velocity magnitude, the isovalue matches the one of Figure 4.3.

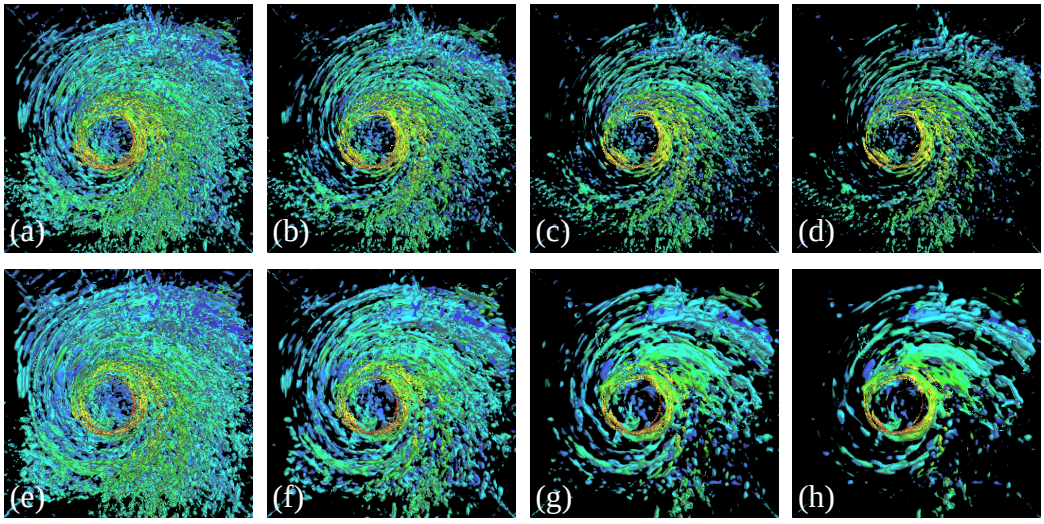

Figure 5.1 λ_2 isosurfaces of the hurricane Isabel data set. Images **(a)–(d)** show the effect of successively decreasing the isovalue, images **(e)–(h)** illustrate the effect of applying various filters starting with the same base image. While both adjustments tend to remove small-scale vortex structures, it can be seen that only filtering is capable of extracting relevant, large-scale vortex structures.

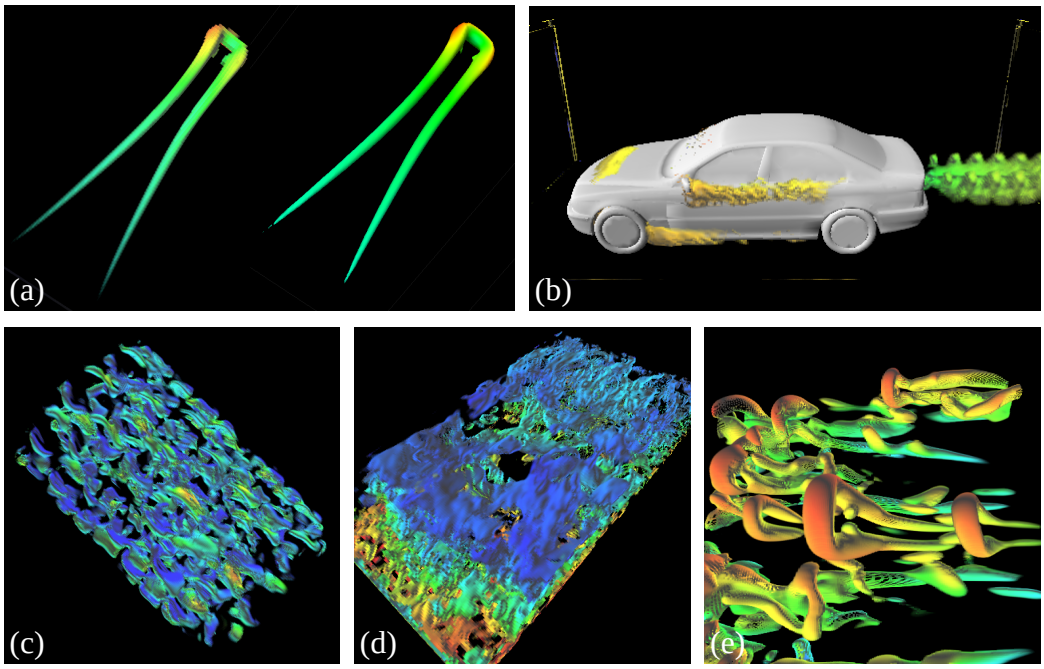

Figure 5.4 Vortices extracted and visualized with the GPU-based implementation of the λ_2 criterion. **(a)** Isolated Λ-vortex; **(b)** flow around car body (resampled grid); **(c)** flow through sphere filling; **(d)** measured water channel flow; **(e)** K-type transition experiments. Data sets courtesy IAG (University of Stuttgart), BMW AG, and LSTM (University of Erlangen).

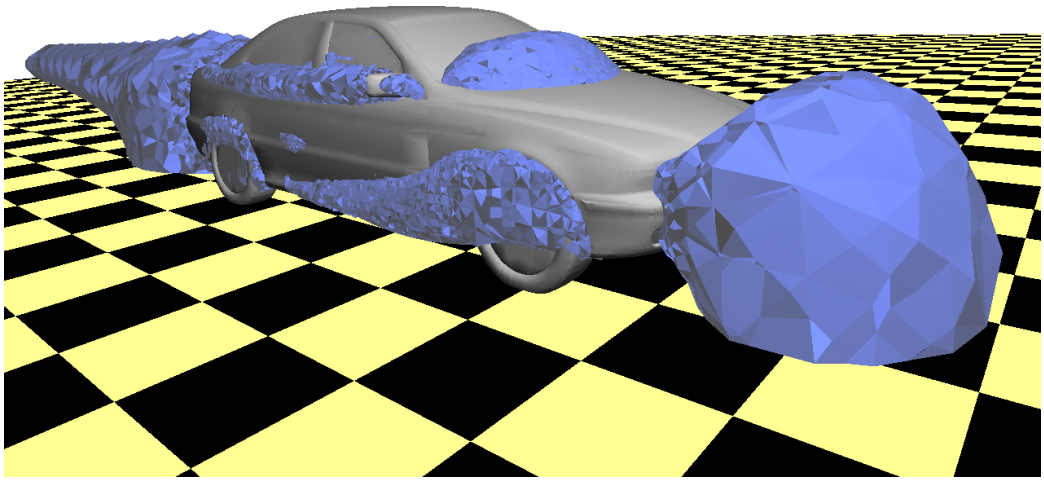

Figure 5.7 Isosurface geometry extracted on the GPU. The scalar property is velocity magnitude, the data is defined on an unstructured grid. Data set courtesy BMW AG.

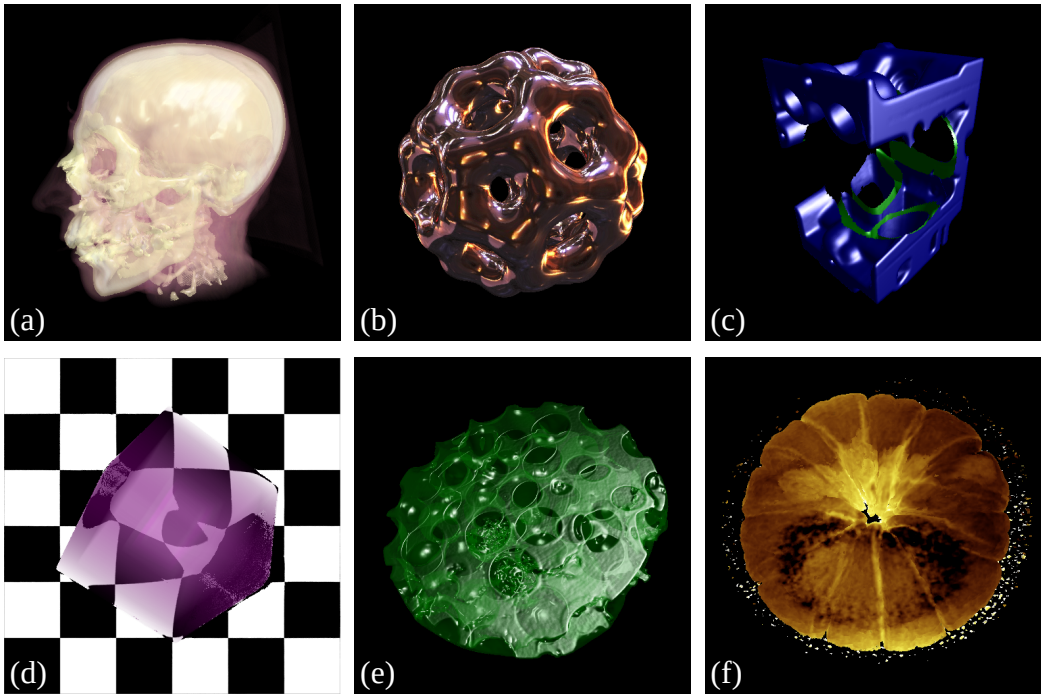

Figure 5.12 Various visualization techniques demonstrating the flexibility of GPU-based ray-casting. **(a)** Combination of isosurfaces and direct volume rendering; **(b)** image-based lighting; **(c)** volume clipping; **(d)** continuous refraction; **(e)** transparent isosurfaces; **(f)** translucency. Data sets courtesy UNC Chapel Hill, UC Davis, General Electric Corporation, University of Erlangen, and Lawrence Berkeley Laboratory.

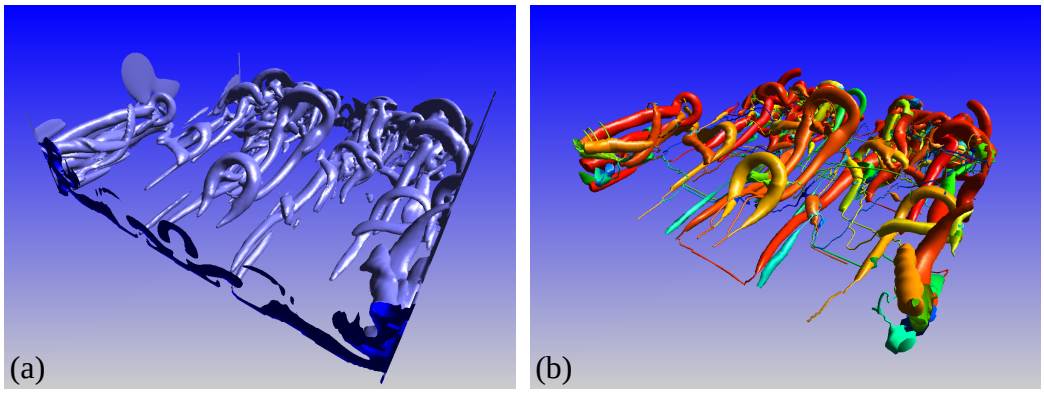

Figure 5.15, Part I Comparison of λ_2 isosurface and vortex structures detected and separated by combining the λ_2 criterion with a predictor-corrector vortex detection method. **(a)** λ_2 isosurface, **(b)** vortices separated with the hybrid approach. *. . . to be continued on page 176.*

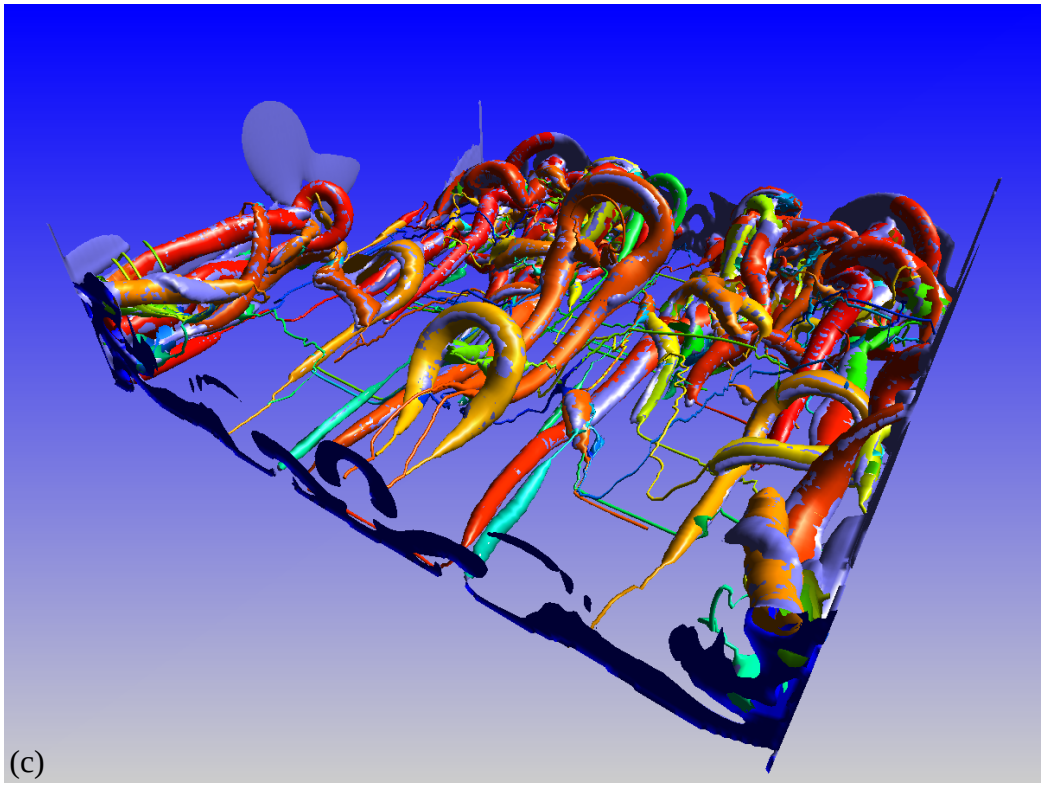

Figure 5.15, Part II *Continued from Figure 5.15, Part I, page 175 . . .* Image **(c)** shows both the isosurface and the separated vortices.

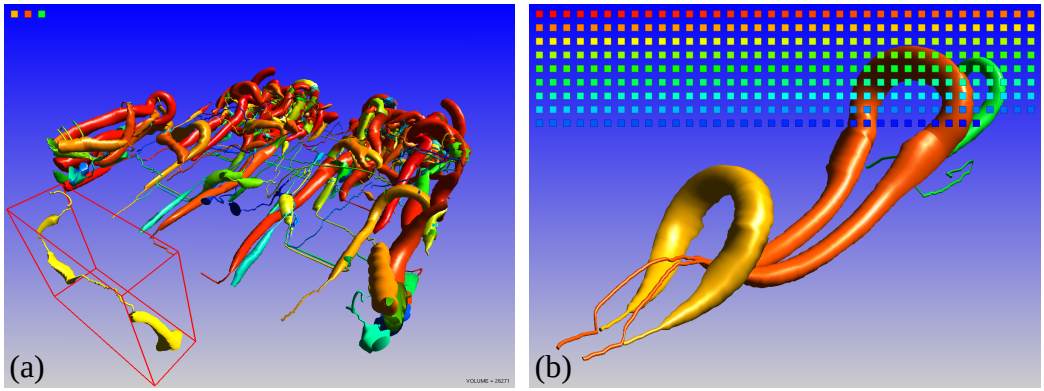

Figure 5.17 Manipulating the vortex set. **(a)** Hiding the three central Λ-vortices and selecting a single vortex for closer examination. **(b)** Inverting the selection. Using the colored buttons individual hidden vortices can be temporarily or permanently re-inserted.

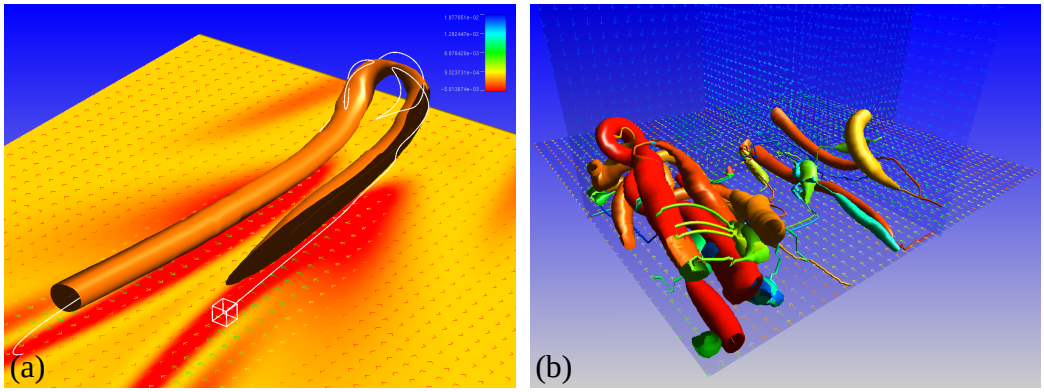

Figure 5.18 (a) Vorticity line traced from near the vortex core. The necessity for a correction step in tracing the core is clearly seen at the Ω-shaped rear part. **(b)** Clipping and vector plot (hedgehog) visualization.

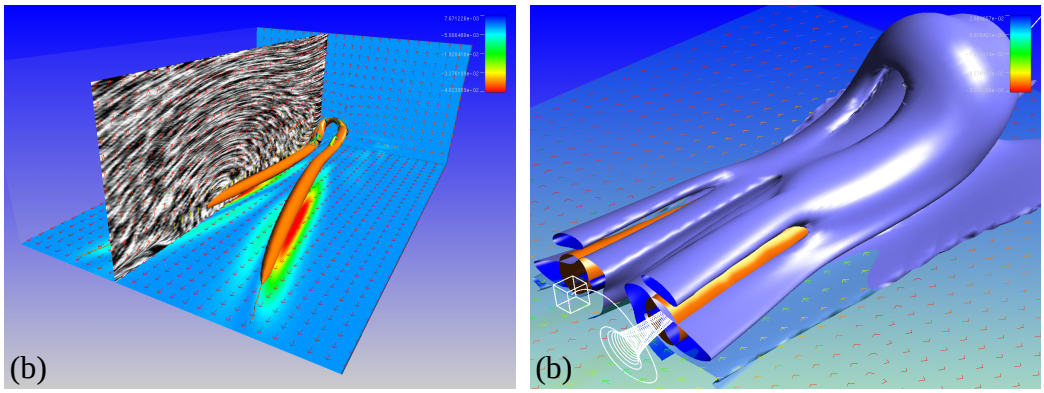

Figure 5.19 (a) Dense vector field representation with LIC and color encoding of scalar field (shear stress), **(b)** Combined visualization of shear layers, cutting planes, arrow plots, and particle traces.

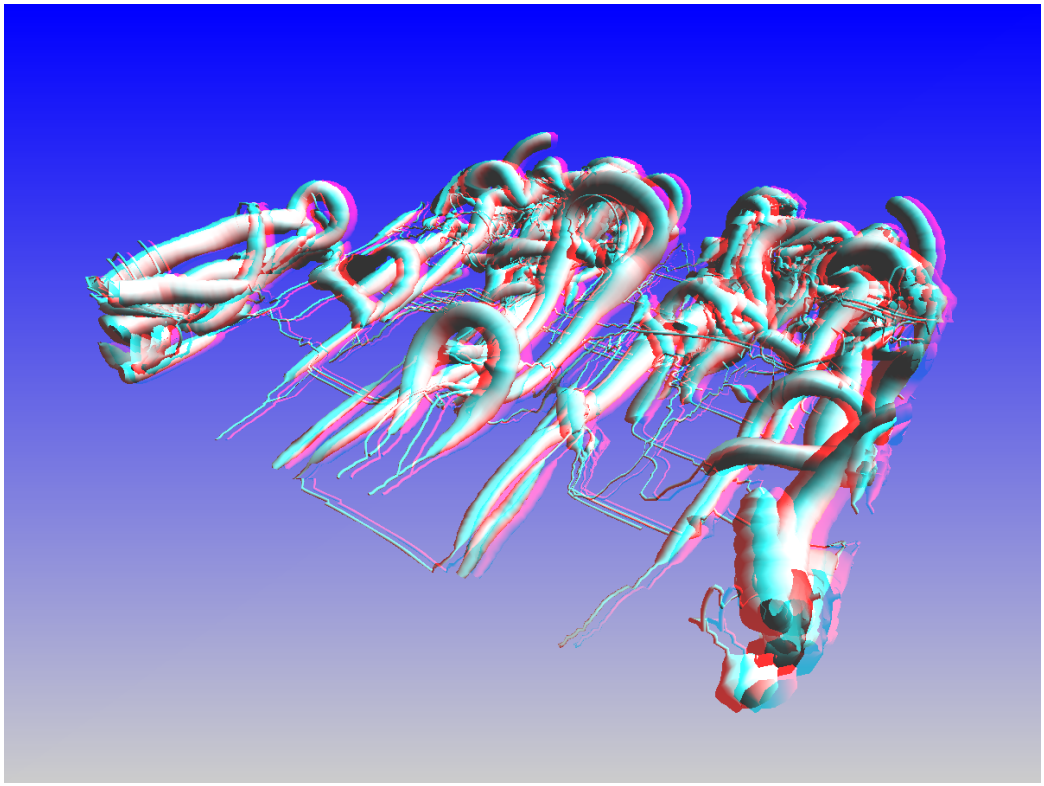

Figure 5.20 Red/cyan anaglyph visualization of separated vortices.

Bibliography

[1] M. J. Aftosmis and M. J. Berger. Multilevel Error Estimation and Adaptive h-Refinement for Cartesian Meshes with Embedded Boundaries. 40th AIAA Aerospace Sciences Meeting and Exhibit, Paper 2002-0863. AIAA, 2002.

[2] B. Akka. Writing Stereoscopic Software for StereoGraphics Systems using Microsoft Windows OpenGL, 1998. StereoGraphics Corporation.

[3] M. Al-Atabi, S. B. Chin, and X. Y. Luo. Visualization of Flow in Circular Tube with Segmental Baffles at Low Reynolds Numbers. In *Electronic Proceedings 11th International Symposium on Flow Visualization '04*, Paper 16, 2004.

[4] M. Bailey and D. Clark. Using OpenGL and ChromaDepth to obtain Inexpensive Single–image Stereovision for Scientific Visualization. *Journal of Graphics Tools*, 3(3):1–9, 1998.

[5] D. C. Banks and B. A. Singer. Vortex Tubes in Turbulent Flows: Identification, Representation, Reconstruction. In *Proceedings of IEEE Visualization '94*, pages 132–139. IEEE, 1994.

[6] U. Behrens and R. Ratering. Adding Shadows to a Texture-based Volume Renderer. In *Proceedings of IEEE Symposium on Volume Visualization '98*, pages 39–46. ACM Press, 1998.

[7] W. Bethel. Visualization Dot Com. *Computer Graphics and Applications*, 20(3):17–20, 2000.

[8] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of SIGGRAPH '93*, pages 263–270. ACM Press, 1993.

[9] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, and L. A. Leske. Two Bit/Pixel Full Color Encoding. In *Proceedings of SIGGRAPH '86*, pages 215–223. ACM Press, 1986.

[10] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[11] W. Citrin, P. Hamill, M. D. Gross, and A. Warmack. Support for Mobile Pen-Based Applications. In *Proceedings of MOBICOM '97*, pages 241–247. ACM, 1997.

[12] D. Commander. VirtualGL—3D Without Boundaries Webpage. http://virtualgl.sourceforge.net, 2004.

[13] N. G. Deen, B. H. Hjertager, and T. Solberg. Comparison of PIV and LDA Measurement Methods Applied to the Gas-Liquid Flow in a Bubble Column. In *10th International Symposium on Applications of Laser Techniques to Fluid Mechanics*, Paper 38.5, 2000.

[14] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans Commun. Elec. Inf. Syst.*, E-74(1):214–224, 1991.

[15] G. Eckel. *OpenGL Volumizer Programmer's Guide*. Silicon Graphics, Inc., Mountain View, 1998.

[16] D. Ellsworth, B. Green, and P. Moran. Interactive Terascale Particle Visualization. In *Proceedings of IEEE Visualization '04*, pages 353–360. IEEE, 2004.

[17] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications. In *Procceedings of IEEE Visualization '00*, pages 449–452. IEEE, 2000.

[18] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *EG/SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, 2001.

[19] K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 167–177, 291. EG/IEEE, 2000.

[20] J. D. Faires and R. L. Burden. *Numerische Methoden*. Spektrum Akademischer Verlag, Heidelberg, 1994.

[21] V. M. Fernandez, N. J. Zabusky, S. Bhat, D. Silver, and S.-Y. Chen. Visualization and Feature Extraction in Isotropic Navier-Stokes Turbulence. In *Proceedings of the AVS95 Conference*, 1995.

[22] R. Fernando, editor. *GPU Gems*. Addison-Wesley Professional, Boston, 2004.

[23] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, Reading, 1997.

[24] S. F. Frisken and R. N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 3(3):1–9, 2002.

[25] J. Fulton and C. K. Kantarjiev. An Update on Low Bandwidth X (LBX). In *The X Resource*, number 5, pages 251–266, 1993.

[26] J.-L. Gailly and M. Adler. ZLIB Library Information Webpage. http://www.gzip.org/zlib, 2005.

[27] R. Grzeszczuk, C. Henn, and R. Yagel. Advanced Geometric Techniques for Ray Casting Volumes. ACM SIGGRAPH 1998 Course #4 Notes, 1998.

[28] H. Hagen, H. Müller, and G. M. Nielson, editors. *Focus on Scientific Visualization*. Springer Verlag, Berlin, 1993.

[29] R. Haimes and D. Kenwright. On the Velocity Gradient Tensor and Fluid Feature Extraction. AIAA 14th Computational Fluid Dynamics Conference, Paper 99-3288. AIAA, 1999.

[30] Hewlett-Packard Development Company, LP. Advantages and Implementation of HP Remote Graphics Software White Paper, 2004. http://www.hp.com/workstations/white_papers/docs/hp_remotegraphics.pdf.

[31] W. W. Ho and R. A. Olsson. An Approach to Genuine Dynamic Linking. *Software, Practice and Experience*, 21(4):375–390, 1991.

[32] R. Hooke and T. A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *J. ACM*, 8(2):212–229, 1961.

[33] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. http://download.intel.com/design/Pentium4/manuals/25366616.pdf, 2005.

[34] J. Jeong and F. Hussain. On the Identification of a Vortex. *Journal of Fluid Mechanics*, pages 69–94, 285 1995.

[35] M. Jiang, R. Machiraju, and D. Thompson. Detection and Visualization of Vortices. In C. Johnson and C. Hansen, editors, *Visualization Handbook*, pages 295–309. Elsevier Academic Press, 2005.

[36] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language. Language Version 1.10, 3DLabs, Inc. Ltd., 2004.

[37] M. J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley Publishing Company, Reading, 1996.

[38] M. J. Kilgard. The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. http://www.opengl.org/documentation/specs/glut/glut-3.spec.pdf, 1996.

[39] M. J. Kilgard, D. Blythe, and D. Hohn. System Support for OpenGL Direct Rendering. In W. A. Davis and P. Prusinkiewicz, editors, *Proceedings of Graphics Interface '95*, pages 116–127. Canadian Human-Computer Communications Society, 1995.

[40] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[41] J. Krahn and F. Villanustre. Stereo3D Library for OpenGL Webpage. http://sourceforge.net/projects/stereogl/, 2001.

[42] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03*, pages 287–292. IEEE, 2003.

[43] M. Lang, U. Rist, and S. Wagner. Investigations on Disturbance Amplification in a Laminar Separation Bubble by Means of LDA and PIV. In *Electronic Proceedings 11th International Symposium on Laser Techniques to Fluid Mechanics*, 2002.

[44] M. Langbein, G. Scheuermann, and X. Tricoche. An Efficient Point Location Method for Visualization in Large Unstructured Grids. In *Workshop on Vision, Modeling, and Visualization VMV '03*, pages 27–35. infix, 2003.

[45] R. S. Laramee. FIRST: A Flexible and Interactive Resampling Tool for CFD Simulation Data. *Computers & Graphics*, 27(6):905–916, 2003.

[46] L. Lipton. StereoGraphics Developers' Handbook. StereoGraphics Corporation, 1997.

[47] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of SIGGRAPH '87*, pages 163–169. IEEE, 1987.

[48] H. Lu. Executable and Linkable Format (ELF) Specification V1.1. ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz, 1995.

[49] K.-L. Ma and D. M. Camp. High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network Status. In *Electronic Proceedings of Supercomputing '00*, Paper 29, 2000.

[50] R. Mace. OpenGL ARB Superbuffers Game Developers Conference 2004 Presentation. http://www.ati.com/developer/gdc/SuperBuffers.pdf, 2004.

[51] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[52] D. Meyer. Direkte numerische Simulation nichtlinearer Transitionsmechanismen in der Strömungsgrenzschicht einer ebenen Platte. PhD Thesis, University of Stuttgart, 2003.

[53] Microsoft Corporation. DirectX 9 SDK Product Information Webpage. http://www.microsoft.com/directx, 2004.

[54] P. Moin and J. Kim. The Structure of the Vorticity Field in Turbulent Channel Flow. Part 1. Analysis of Instantaneous Fields and Statistical Correlations. *Journal of Fluid Mechanics*, page 441, 155 1985.

[55] R. W. D. Nickalls. A New Approach to Solving the Cubic: Cardan's Solution Revealed. In *Mathematical Gazette*, volume 77, pages 354–359. The Mathematical Association, 1993.

[56] R. W. D. Nickalls. Solving the Cubic Using Tables. *Theta*, 10(2):21–24, 1996.

[57] NVIDIA Corporation. Cg Toolkit User's Manual. Version 1.5, 2002.

[58] NVIDIA Corporation. NVIDIA OpenGL Extension Specifications Webpage. http://developer.nvidia.com/object/nvidia_opengl_specs.html, 2004.

[59] NVIDIA Corporation. NVIDIA SDK 8.0 Product Information Webpage. http://developer.nvidia.com/object/sdk_home.html, 2004.

[60] A. Nye. *Volume 0: X Protocol Reference Manual*. X Window System Series. O'Reilly & Associates, Sebastopol, 4th edition, 1995.

[61] A. Nye. *Volume 1: Xlib Programming Manual*. X Window System Series. O'Reilly & Associates, Sebastopol, 3rd edition, 1995.

[62] M. Oberhumer. LZO Real-Time Data Compression Library Webpage. http://www.oberhumer.com/opensource/lzo, 2005.

[63] J.-R. Ohm. *Digitale Bildcodierung*. Springer Verlag, Berlin, 1995.

[64] V. Pascucci. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Proceedings of EG/IEEE TVCG Symposium on Visualization VisSym '04*, pages 293–300. EG/IEEE, 2004.

[65] C. Peeper. DirectX High Level Shading Language. Microsoft Meltdown UK Presentation, Microsoft Corporation, 2002.

[66] R. Peikert and M. Roth. The "Parallel Vectors" Operator – A Vector Field Visualization Primitive. In *Proceedings of IEEE Visualization '99*, pages 263–270. IEEE, 1999.

[67] D. Poirier, S. R. Allmaras, D. R. McCarthy, M. F. Smith, and F. Y. Enomoto. The CGNS System. AIAA 29th Fluid Dynamics Conference, Paper 98-3007. AIAA, 1998.

[68] L. M. Portela. Identification and Characterization of Vortices in the Turbulent Boundary Layer. PhD Thesis, Stanford University, 1997.

[69] F. H. Post and T. van Walsum. Fluid Flow Visualization. In *Focus on Scientific Visualization*, pages 1–40, Berlin, 1993. Springer Verlag.

[70] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The State of the Art in Flow Visualisation: Feature Extraction and Tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.

[71] D. Qian. Bubble Motion, Deformation, and Breakup in Stirred Tanks. PhD Thesis, Clarkson University, 2003.

[72] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *EG/SIGGRAPH Workshop on Graphics Hardware '00*, Annual Conference Series, pages 109–118,147. Addison-Wesley Publishing Company, 2000.

[73] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[74] U. Rist, K. Müller, and S. Wagner. Visualization of Late-Stage Transitional Structures in Numerical Data using Vortex Identification and Feature Extraction. In *Electronic Proceedings 8th International Symposium on Flow Visualization '98*, Paper 103, 1998.

[75] J. A. Robinson. Efficient General-Purpose Image Compression with Binary Tree Predictive Coding. *IEEE Transactions on Image Processing '97*, 6(4):601–607, 1997.

[76] S. K. Robinson. Coherent Motions in the Turbulent Boundary Layer. *Annu. Rev. Fluid Mech*, 23:601–639, 1991.

[77] D. Rose, S. Stegmaier, G. Reina, D. Weiskopf, and T. Ertl. Non-invasive Adaptation of Black-box User Interfaces. In *Proceedings of Fourth Australasian User Interface Conference AUIC 2003*, pages 19–24, 2003.

[78] M. Roth and R. Peikert. A Higher-Order Method for Finding Vortex Core Lines. In *Proceedings of IEEE Visualization '98*, pages 143–150. IEEE, 1998.

[79] S. Röttger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238. EG/IEEE, 2003.

[80] A. Sadarjoen, T. van Walsum, A. J. S. Hin, and F. H. Post. Particle Tracing Algorithms for 3D Curvilinear Grids. In *Proceedings 5th Eurographics Workshop on Visualization in Scientific Computing*, 1994.

[81] J. Sahner, T. Weinkauf, and H.-C. Hege. Galilean Invariant Extraction and Iconic Representation of Vortex Core Lines. In *Proceedings Eurographics/IEEE VGTC Symposium on Visualization EuroVis '05*, pages 151–160, 2005.

[82] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, Reading, 1990.

[83] M. Schäfer. *Numerik im Maschinenbau*. Springer Verlag, Berlin, 1999.

[84] M. Schulz, F. Reck, W. Bartelheimer, and T. Ertl. Interactive Visualization of Fluid Dynamics Simulations in Locally Refined Cartesian Grids. In *Procceedings of IEEE Visualization '99*, pages 413–416. IEEE, 1999.

[85] Science+Computing AG. PowerVIZ Product Information Webpage. http://www.science-computing.de/software/powerviz.html, 2005.

[86] Silicon Graphics, Inc. OpenGL Vizserver 3.0—Application-Transparent Remote Interactive Visualization and Collaboration. http://www.sgi.com, 2003.

[87] D. Silver and N. Zabusky. Quantifying Visualizations for Reduced Modeling in Nonlinear Science: Extracting Structures from Data Sets. *J. Visual Comm. and Image Representation*, 4(1):46–61, 1993.

[88] I. Sobel. An Isotropic $3\times3\times3$ Volume Gradient Operator. Unpublished manuscript, 1995.

[89] D. Speray and S. Kennon. Volume Probes: Interactive Data Exploration on Arbitrary Grids. In *Proceedings of Workshop on Volume Visualization '90*, pages 5–12. ACM Press, 1990.

[90] D. Stalling and H.-C. Hege. Fast and Resolution-Independent Line Integral Convolution. In *Proceedings of SIGGRAPH '95*, pages 249–256. ACM Press, 1995.

[91] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. Widening the Remote Visualization Bottleneck. In *Proceedings of IEEE International Symposium on Image and Sugnal Processing and Analysis '03*, pages 174–179. IEEE, 2003.

[92] S. Stegmaier and T. Ertl. A Graphics Hardware-based Vortex Detection and Visualization System. In *Proceedings of IEEE Visualization '04*, pages 195–202. IEEE, 2004.

[93] S. Stegmaier and T. Ertl. On a Graphics Hardware-based Vortex Detection and Visualization System. In *Electronic Proceedings 11th International Symposium on Flow Visualization '04*, Paper 85, 2004.

[94] S. Stegmaier and T. Ertl. On a Graphics Hardware-based Vortex Detection and Visualization System. *Journal of Visualization*, 8(2):153–160, 2005.

[95] S. Stegmaier, M. Magallón, and T. Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '02*, pages 87–94. EG/IEEE, 2002.

[96] S. Stegmaier, U. Rist, and T. Ertl. Opening the Can of Worms: An Exploration Tool for Vortical Flows. In *Proceedings of IEEE Visualization '05*, pages 463–470. IEEE, 2005. *to appear*.

[97] S. Stegmaier, D. Rose, and T. Ertl. A Case Study on the Applications of a Generic Library for Low-Cost Polychromatic Passive Stereo. In *Procceedings of IEEE Visualization '02*, pages 557–560. IEEE, 2002.

[98] S. Stegmaier, M. Schulz, and T. Ertl. Resampling of Large Datasets for Industrial Flow Visualization. In *Workshop on Vision, Modeling, and Visualization VMV '03*, pages 375–382. infix, 2003.

[99] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of Volume Graphics '05*, pages 187–193,241, 2005.

[100] D. Sujudi and R. Haimes. Identification of Swirling Flow in 3D Vector Fields. AIAA 12th Computational Fluid Dynamics Conference, Paper 95-1715. AIAA, 1995.

[101] C. Teitzel, R. Grosso, and T. Ertl. Efficient and Reliable Integration Methods for Particle Tracing in Unsteady Flows on Discrete Meshes. In *Proceedings 8th Eurographics Workshop on Visualization in Scientific Computing*, pages 49–56, 1997.

[102] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley Professional, Boston, 1992.

[103] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, and K. I. Joy. Extraction of Crack-free Isosurfaces from Adaptive Mesh Refinement Data. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 25–34, 335. EG/IEEE, 2001.

[104] M. Weiler, R. P. Botchen, J. Huang, Y. Jang, S. Stegmaier, K. P. Gaither, D. S. Ebert, and T. Ertl. Hardware-assisted Feature Analysis and Visualization of Procedurally Encoded Multifield Volumetric Data. *Computer Graphics and Applications*, pages ???–???, 2005. *to appear*.

[105] M. Weiler and T. Ertl. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. In *Procceedings of IEEE Visualization '01*, pages 199–206. IEEE, 2001.

[106] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Procceedings of IEEE Visualization '03*, pages 333–340. IEEE, 2003.

[107] D. Weiskopf and G. Erlebacher. Overview of Flow Visualization. In C. Johnson and C. Hansen, editors, *Visualization Handbook*, pages 261–278. Elsevier Academic Press, 2005.

[108] R. Westermann. The Rendering of Unstructured Grids Revisited. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 65–74. EG/IEEE, 2001.

[109] M. Woo, J. Neider, and T. David. *OpenGL 1.2 Programming Guide, 3rd Edition: The Official Guide to learning OpenGL, Version 1.2*. Addison-Wesley Publishing Company, Reading, 1999.

[110] J.-Z. Wu, A.-K. Xiong, and Y.-T. Yang. Axial Stretching and Vortex Definition. *Phys. Fluids*, 17(3):038108–1–4, 2005.