Exploiting Programmable Graphics Hardware for Interactive Visualization of 3D Data Fields

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

Thomas Klein

aus Bietigheim

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. D. Weiskopf
Mitberichter:	Prof. Dr. D. Ebert

Tag der mündlichen Prüfung: 16. Oktober 2008

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2008

Contents

List of Abbreviations and Acronyms

ACM	Association for Computing Machinery	GPGPU	General-Purpose computation on GPUs
ALU	Arithmetic Logic Unit	GPU	Graphics Processing Unit
API	Application Programming Interface	GHz	gigahertz
ARB	OpenGL Architecture Review Board	HLSL	High Level Shading Language
bit	binary digit	IEEE	Institute of Electrical and Electronics Engineers
byte	eight bits	i.e.	id est
CFD	Computational Fluid Dynamics	LOD	level of detail
Cg	C for Graphics	MB	megabyte
CIE	Commission internationale de l'clairage	MIP	maximum intensity projection
CPU	Central Processing Unit	MR	Magnetic Resonance
CT	Computer Tomography	MRI	Magnetic Resonance Imaging
CUDA	Compute Unified Device Architecture	NIH	National Institutes of Health
dB	decibel	NSF	National Science Foundation
DT-MRI	Diffusion Tensor Magnetic Resonance Imaging	OpenCL	Open Computing Language
e.g.	exempli gratia	OpenGL	Open Graphics Library
et al.	et alii	PC	Personal Computer
etc.	et cetera	PCIe	Peripheral Component Interconnect Express
FEM	Finite element method	RAM	Random Access Memory
FPGA	Field-programmable Gate Array	RAMDAC	RAM Digital-to-Analog Converter
fps	frames per second	RGB	red, green, blue
FSAA	full-scene antialiasing	RGBA	red, green, blue, alpha
GB	gigabyte	SIMD	single instruction, multiple data
GFlops	billions of floating point operations per second	SPMD	single program, multiple data
GLSL	OpenGL Shading Language	TFlops	trillions of floating point operations per second

Abstract

Modern numerical simulation and data acquisition techniques create a multitude of different data fields. The interactive visualization of these large, three-dimensional, and often also time-dependent scalar, vector, and tensor fields plays an integral part in analysing and understanding this data. Although basic visualization techniques vary significantly depending on the type of the respective data fields, there is one key feature that is dominating in today's visualization research. Driven by the need for interactive data inspection and exploration and by the extraordinary rate of increase of the computational power provided by modern graphics processing units, the attempt for consequent application of graphics hardware in all stages of the visualization pipeline has become a central theme in order to cope with the challenges of data set sizes growing at an ever increasing pace and advancing demands on the accuracy and complexity of visualizations. Contemporary graphics processing units now have reached a level of programmability roughly resembling their CPU counterparts. However, there are still important differences that strongly influence the design and implementation of GPU-based visualization algorithms.

This thesis addresses the problem of how to efficiently exploit the programmability and parallel processing capabilities of modern graphics processors for interactive visualization of three-dimensional data fields of varying data complexity and abstraction level. In particular new methods and GPU-based solutions for high-quality volume ray casting, the reconstruction of polygonal isosurfaces, and the point-based visualization of symmetric, second-order tensor fields, such as obtained by diffusion tensor imaging or resulting from CFD simulations, by means of ellipsoidal glyphs are presented that by combining the mapping and rendering stage onto the GPU result in an improved visualization cycle. Furthermore, a new approach for the topological analysis of noisy vector fields is described.

Although this work is focused on a number of specific visualization problems, it also intends to identify general design principles for GPU-based visualization algorithms that may prove useful in the context of topics not covered by this thesis.

Zusammenfassung

Moderne Verfahren der numerische Simulation und der experimentellen Datenerfassung erzeugen eine Vielzahl unterschiedlicher Datenfelder. Die Visualisierung großer, dreidimensionaler und in vielen Fällen zeitabhängiger Skalar-, Vektor- und Tensorfelder spielt eine zentrale Rolle für die Auswertung und das Verständnis dieser Daten. Wenngleich die grundlegenden Visualisierungstechniken für die oben genannten Felder sich wesentlich unterscheiden haben sie doch eine Aspekt gemeinsam, welcher die heutige Forschung im Bereich der Visualisierung entscheidend bestimmt. Getrieben durch den Wunsch nach interaktiver Darstellung und Exploration der Daten und begünstigt durch den außerordentlichen Zuwachs an Rechenleistung, die moderne Graphikprozessoren heute zur Verfügung stellen, hat sich der möglichst umfassende Einsatz von Graphik-Hardware in allen Stufen der Visualisierungspipeline als zentraler Ansatz herauskristallisiert, um den Herausforderungen, welche durch die immer schneller ansteigenden Datensatzgrößen und den ebenfalls stetig wachsenden Anforderungen an die Genauigkeit und die Komplexität der Visualisierungsaufgaben auf die Visualisierung zukommen, gerecht zu werden.

Die vorliegende Arbeit beschäftigt sich daher mit dem effizienten Einsatz aktueller programmierbarer Graphik-Hardware und der Ausnutzung der ihr eigenen Parallelität zur interaktiven Visualisierung dreidimensionaler Datenfelder. Dabei werden Felder unterschiedlicher Datenkomplexität und verschiedener Abstraktionsebenen betrachtet. Insbesondere werden neue Methoden und GPU-basierte Ansätze für hochqualitatives Volumen-Raycasting, die Rekonstruktion polygonaler Isoflächen und eine punktbasierte Visualisierungsmethode für symmetrische Tensorfelder zweiter Ordnung beschrieben, welche den Visualisierungsprozess, durch entsprechende Kombination des Mapping- und Renderingschritts der Visualisierungspipeline in einem GPU-basierten Algorithmus, deutlich verbessern. Außerdem wird ein neuer Ansatz zur topologischen Analyse verrauschter Vektorfelder vorgestellt.

Obwohl sich diese Arbeit im wesentlichen auf einige spezielle Visualisierungsprobleme konzentriert, sollen die vorgestellten Methoden auch dazu dienen allgemeine Entwurfsprinzipien für GPU-basierte Visualisierungsalgorithm zu identifizieren, die sich auch außerhalb des durch die Arbeit gesteckten Rahmens als hilfreich erweisen können.

Chapter 1

Introduction

Visualization plays a key role in today's engineering and research landscape and has been established as a central tool in the postprocessing pipeline in many scientific, engineering, and medical disciplines. The visual analysis of simulation results or measurement data provides the means of interpretation of that data and facilitates the understanding necessary for gaining insight into the raw numerical values.

Recent advances in simulation and acquisition technology are making available an unprecedented amount of data. The total aggregate supercomputing power of the systems represented by the Top 500 list of high-performance computers[I] has followed Moore's Law very closely, roughly doubling in value every year. This same trend is also clearly visible for small and medium-sized simulation systems that are widespread used in engineering and research. Processing power is steadily growing and accordingly simulations are carried out at increasing levels of detail. Thus, growing problem sizes are producing ever increasing amounts of simulation data. While few years ago 2D approximations were commonly employed to contain the computational complexity of numerical simulation, nowadays large three-dimensional simulations, for example in computational fluid dynamics, are ubiquitous in industry and academia. At the same time the increased resolution and availability of high-precision data acquisition devices, such as computed tomography scanners or magnetic resonance imaging devices, present another source of large amounts of data.

Hence, new challenges have been posed for visualization research to provide practitioners with effective and efficient visualization software that allows real-time interaction with and interactive exploration of these data sets. In particular, interactive control of the visualization is of utmost importance when dealing with large three-dimensional data. The often very complex three-dimensional struc-

[I] http://www.top500.org

ture of the visualized phenomena requires the user of the visualization tool to cope with the problems of occlusion and scene complexity by inspecting the data from different angles and with different visualization parameters. This demand for interactivity has to be addressed in different ways: It is not only necessary to improve existing algorithms, but also to develop new visual representations for complex data fields as well as new algorithms that reduce the visual complexity and the amount of information presented to the viewer.

One solution to improve existing algorithms in this regard is exploiting parallelism. However, while simulations are commonly carried out on supercomputers, data analysis and visualization often has to make do with the limited resources provided by workstations or desktop computers. On the other hand, commodity graphics hardware, which is found today in every modern desktop computer, provides computational and graphics processing capabilities that already exceed the power of state-of-the-art CPUs by an order of magnitude. This is possible since standard graphics processing units are based on a massively parallel computational model featuring arrays of several hundreds of processor cores. Even more, graphics processing units are still evolving very fast and their increase in peak computational power is growing faster than that of CPUs, surpassing Moore's Law by doubling the number of transistor functions approximately every six months. In his comments on the state of scientific visualization research [90] Johnson identifies the need for efficient utilization of novel hardware architectures, like programmable graphics processing units, and to harness this cheap and widely available resource for scientific visualization as one of the top visualization research challenges. This notion has been also picked up in the NIH/NSF Visualization Research Challenges Report [91] in 2006 and the impact that readily available commodity graphics hardware already had on the progress in visualization has been emphasized.

1.1 Problem Statement and Thesis Overview

This thesis addresses the problem of how to efficiently exploit the programmability and parallel processing capabilities of modern commodity graphics processors for interactive visualization of three-dimensional data fields of varying data complexity and abstraction level. There is no general approach for dealing with scalar-valued volumetric data sets, vector fields, and tensor fields. Each field type poses its own challenges to GPU-based visualization and requires specific application dependent solutions and visualization techniques. Therefore, a number of different directions of how to tackle the peculiarities of the respective fields have been explored in this thesis.

In particular new methods and GPU-based solutions for visualizing scalar vol-

umes by high-quality volume ray casting and reconstruction of polygonal isosurfaces, the visualization of symmetric, second-order tensor field data and vector fields by means of point-based quadric glyphs, and the topological analysis of noisy vector fields are presented.

The remainder of this thesis is organized as follows: The following two chapters provide an introduction to programmable graphics hardware (Chapter 2) and introduce fundamental visualization concepts (Chapter 3), providing the necessary background information for the forthcoming chapters. Besides a presentation of important features and recent advances in programmability of modern graphics processing units, basic visualization nomenclature is introduced, a classification of visualization techniques is provided, and the structure of the (GPU-based) visualization pipeline is discussed.

In Chapter 4, the basics of direct volume rendering are introduced and an approach for single-pass GPU volume ray casting is presented. The simplicity and flexibility of this approach is highlighted by the discussion of how a number of acceleration techniques known from traditional volume ray casting, such as early ray termination, adaptive sampling, or empty space skipping, and various optical models, including an extension towards photorealistic volume rendering of refractive volumes, can be realized. Furthermore, a generic volume visualization framework is described. This framework is also used to evaluate GPU-based ray casting in terms of image fidelity and rendering performance and to compare it with traditional slice-based volume rendering.

Chapter 5 discusses a technique for the GPU-based reconstruction of polygonal isosurfaces. In contrast to the ray casting approach presented in the preceding chapter, which allows to visualize isosurfaces without actually reconstructing a surface, this technique reconstructs polygonal geometry that can be further processed directly on the GPU. Furthermore, this approach is capable of extracting isosurfaces from arbitrary unstructured (also nonconvex) grids.

While for scalar-valued volumetric fields natural and descriptive metaphors, like direct volume rendering or isosurface representations, exist, there are no such direct physically motivated representations for higher-order and multi-variate data, like vector and tensor fields. Typical approaches reduce the dimensionality of the problem by either computing derived quantities and visualizing these fields instead or by extracting and visualizing a suitably defined abstract feature representation. Such an abstract representation is often achieved by probing the field at appropriate positions and mapping its local attributes to various geometrical properties of application specific glyphs. In Chapter 6 a point-based approach for glyph-based visualization of three-dimensional symmetric, second-order tensor fields, such as obtained by diffusion tensor imaging or resulting from fluid dynamics simulations, is presented. Combining the mapping and rendering stage onto the GPU results in an improved visualization cycle that allows to interactively

visualize large numbers of individual glyphs. Furthermore, a novel quadric glyph specifically designed for the visualization of indefinite tensors is introduced. In combination with the ray casting technique introduced in Chapter 4 the glyphs are employed for visualizing multi-field data. Last, the effectiveness of the glyphs is demonstrated using examples from the illustrative rendering of magnetic field simulations and the visualization of tensor fields, both in the medical setting, i.e. in order to visualize data from diffusion tensor imaging, and for the visualization of shear tensor data from computational fluid dynamics simulations.

However, in some cases straight forward visualization is not possible, either because the data is too large for interactive visualization due to limited hardware resources, or the shear amount of information and the complexity of the data will put too much burden onto the user when asked to identify important features from such a visualization. In this case one can employ feature extraction in order to create a compact representation of the data that captures the important properties of the underlying field in a more abstract representation. In case of vector fields the topological analysis is an established technique to show the essential properties and the qualitative structure of the field. In this respect a new approach for the topological analysis of noisy vector fields is presented in Chapter 7.

Although this work is focused on a number of specific visualization problems, it also intends to identify general design principles for GPU-based visualization algorithms that may prove useful in other situations. Therefore, Chapter 8 lists a number of *dos and don'ts* that have to be taken into account in order to leverage the full computational potential of the underlying graphics hardware, and a number of general rules and guidelines on how to map visualization algorithms to the GPU avoiding possible traps and pitfalls involved in this process are presented.

Last, it can be noted that, although in this thesis the problem of dealing with very large data sets, which cannot be fit into the locally available graphics memory, will not be directly addressed, all presented techniques can be easily generalized and implemented in a parallel fashion on a GPU cluster computer.

Chapter 2

Programmable Graphics Hardware

In the last decade computer graphics hardware has undergone a remarkable evolution. Driven by the ever increasing performance demands of todays gaming and edutainment industry, cheap, mass-market graphics hardware has rendered dedicated graphics workstation hardware of former years obsolete long ago. The rapid pace at which graphics processing units (GPUs) have evolved even surpassed Moore's Law. While for CPUs the number of transistor functions in the past has been increased at a rate of roughly a factor of two per year, GPUs have grown at a much higher rate doubling the number of transistor functions approximately every six months. Currently there seems to be no end to this trend. In terms of raw computational power a single high-end GPU is already on an order of magnitude faster than currently available high-end CPU cores. As of September 2008, the theoretical single-precision peak performance of a main-stream NVIDIA GeForce GTX 280 GPU is 933 GFlops compared to the combined 96 GFlops delivered by the four cores of an Intel Core 2 Quad Q9650 processor. Even more, in this simple example, the above mentioned GPU delivers already more than twelve times the computational power per unit money.

Simultaneously the former rather inflexible fixed-function graphics processing pipeline has evolved into a highly flexible, massively parallel, programmable stream-computing pipeline. Originally designed to enable complex shading tasks this also makes it possible to move more and more parts of the visualization pipeline to the GPU as will be shown in this thesis.

Even more, the sole purpose of the GPU for graphics applications has been mitigated by this developments. The metamorphosis of graphics processors into highly optimized multiprocessor cores has opened up possibilities for whole new application areas. Nowadays, GPUs are no longer used solely for rendering but have also found applications as numerical co-processing units in simulation and general stream computing.

In this chapter a brief introduction of the fundamental concepts of the GPU

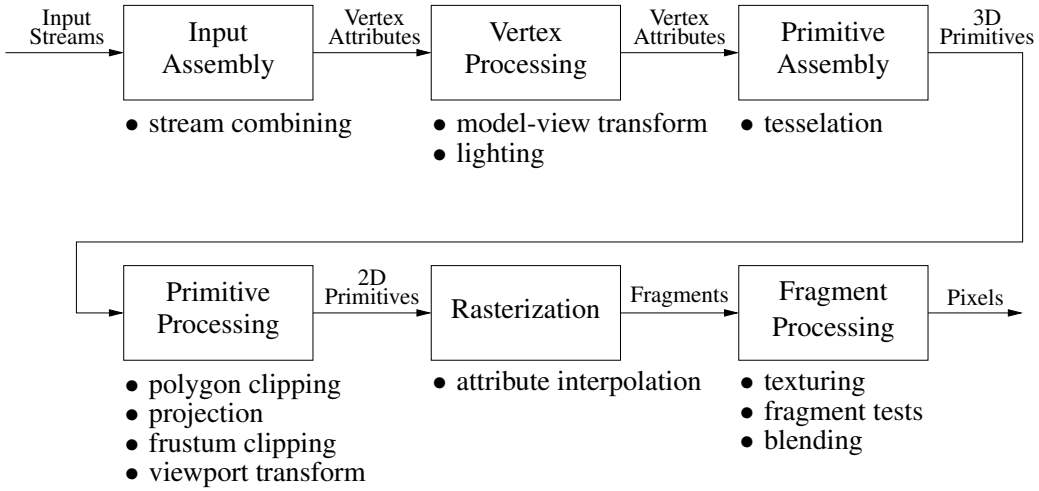


Figure 2.1: Simplified representation of the classical fixed-function rendering pipeline common to all graphics processors prior to the introduction of programmable elements.

rendering pipeline necessary for understanding the forthcoming chapters will be given. Furthermore, in order to understand certain considerations that have been decisive criteria in the design of the visualization techniques presented in this thesis a short review of the evolution of programmable graphics hardware from the classical nonprogrammable, fixed-function pipeline to its current state has to be given. Note that although OpenGL terminology will be used in the following description, all described concepts are not restricted to a specific graphics API.

2.1 Classical Rendering Pipeline

Figure 2.1 depicts the classical rendering pipeline common to all graphics processors before the introduction of programmable shader units. It can be basically split into three parts: the vertex processing pipeline, the primitive pipeline, and the fragment processing pipeline. These are are connected to each other by the primitive assembly and the rasterization stage respectively. This distinction reflects the fundamental change in data modality when moving from three-dimensional geometric data, i.e. vertices and vertex attributes to rasterization primitives, i.e. points, lines, and triangles, and finally to image elements, i.e. fragments and pixels. Common to all parts of the pipeline is the concept of stream processing, i.e. individual data elements, vertices, primitives, and fragments, are processed independently by conceptually parallel working function units.

Input Assembly The first stage of the graphics pipeline is the input assembler. This unit is responsible for combining the different data input streams comprising the scene data, i.e. vertices, vertex normals, vertex colors, texture coordinates, et cetera, into a single vertex attribute stream that is processed by the subsequent vertex processing stage.

Vertex Processing The next part of the rendering pipeline is responsible for transforming three-dimensional vertex attribute data, usually specified in object-space coordinates, into view-space coordinates. First, the stream of homogeneous input vertex coordinates is subject to the modelview transform that combines the modeling transformations, transforming object-space coordinates to world-space coordinates, and the viewing transformation, transforming world space to eye space, into a single affine 4×4 transformation matrix.

Furthermore, per-vertex lighting calculations are performed in view space according to the Phong-Blinn [16] lighting model. Therefore, the vertex processing stage is also commonly referred to as the transform and lighting stage of the rendering pipeline.

Primitive Assembly In order to rasterize the geometrical input primitives to an image the in the vertex processing stage independently treated stream of vertices has to be reassembled into basic primitives supported by the hardware, like points, lines, or triangles, according to the primitive mode specified when the input data streams were issued. This requires also the tesselation of more complex primitive types, such as quadrilaterals, polygons, or triangle strips, into basic primitives.

Primitive Processing The primitive processing unit has several responsibilities. First, primitives generated in the primitive assembly stage have to be clipped against user-defined clip planes using line or polygon clipping. This may remove entire primitives but may also create additional primitives that are to be fed back into the pipeline. In the next step, the remaining primitives are subject to the eye-space-to-screen-space transformation. This nonlinear projection is split in two parts. First, the homogeneous vertex positions are projected onto the image plane according to a 4×4 projection matrix, resulting in clip-space coordinates. This projection matrix defines the so-called view frustum, i.e. the geometrical extent of the visible scene. In most cases a perspective projection is used. After that primitives are clipped against the view frustum, removing those parts of the rendered scene not visible according to the specified view and projection transform.

The second part, perspective division by the homogeneous w-coordinate of the projected vertices, and the following viewport transformation completes the

transition from four-dimensional homogeneous coordinates to two-dimensional image coordinates and an additional depth value.

Rasterization The rasterization stage marks the transition from 3D geometry to 2D fragments in the rendering pipeline. In this stage the projected screen-space primitives are converted into fragments. Thereby, for each generated fragment, vertex attributes are interpolated linearly across the primitive taking the adjacency information obtained during primitive assembly into account.

Fragment Processing After rasterization follow the last functional units of the rendering pipeline. These include texture fetch and application, fragment tests, i.e. scissoring, depth test, alpha test, and stencil test, and framebuffer compositing or alpha-blending. After that, color and depth values for fragments that have passed the fragment tests are written to the framebuffer forming, if they are not over-written by subsequently rasterized fragments, pixels of the final image.

2.2 Evolution of Programmability

Since the first introduction of OpenGL in 1992, there has been an increasingly fast transition from the rigid fixed-function pipeline over a configurable fragment pipeline to a widely programmable rendering pipeline, providing successively more degrees of freedom and programmability for the application developer. In this section the major breakthroughs of this development will be briefly outlined. However, this discussion will be focused on commodity PC and workstation graphics hardware and will, for example, not cover the vast field of video game console graphics hardware.

Configurable Fragment Processing The first step towards a more flexible rendering pipeline was the introduction of configurable fragment processing units to the pipeline. In this context, configurability is meant in a sense that goes beyond mere enabling and disabling of states or changing certain state variables.

Real configurability of the OpenGL rendering pipeline started out with multitexturing and configurable texture environments and culminated in the introduction of texture shaders and register combiners support. The combination of multitexturing and texture shader stages replaces conventional texture mapping with a much more flexible mechanism of a sequence of texture lookup stages, each running one of 21 predefined texture shader functions. These functions include conventional texture mapping but also allow dependent texture lookups, dot product

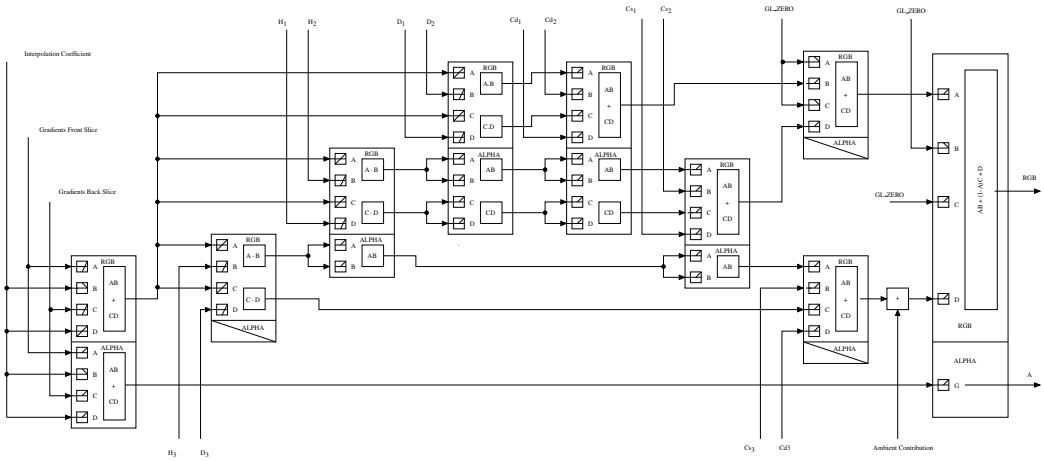


Figure 2.2: Register combiners based lighting of an isosurface using three independent directional light sources.

computations, and other more specialized functions not possible in the standard pipeline.

While the texture shader extension allows for advanced control of texture coordinate computation and texture sampling, register combiners provide a configurable mechanism to control the OpenGL texture environment, color sum, and fog application. Each combiner stage can use up to four inputs, including primary, secondary, and fog color, an additional user-defined constant color, the interpolated texture output from the texture shader, and the output of previous combiner stages, and is capable of computing arithmetic operations, such as additions, multiplications, and dot products to produce the final fragment color.

Figure 2.2 shows an example of a register combiners setup used for isosurface shading in a slice-based volume renderer [216].

Although the combination of texture shaders and register combiners allowed the implementation of a number of at that time astonishing new visual effects in interactive graphics applications, such as bump-mapping, Phong-shading, or refraction [99], and provided additional degrees of freedom to the application developer, their applicability for more general computations, which are essential in moving larger parts of the visualization pipeline to the GPU, was rather limited. Furthermore, only parts of the fragment processing unit were configurable, while vertex and primitive processing was still fixed functionality.

Programmable Vertex and Fragment Processing Real programmability of the rendering pipeline started out with the introduction of programmable ver-

tex processing units that replaced the fixed function vertex processing stage of the pipeline. Instead of relying on the predefined vertex processing the application developer now was free to implement his own vertex attribute transformation pipeline, including full control over modelview transformation, perspective projection, and per-vertex lighting calculations. This programmable vertex processing unit could be programmed using small assembler language programs, in OpenGL terminology so-called vertex programs, with a graphics oriented SIMD instruction set optimized for 4D vertex processing. Program input is provided by a number of predefined input registers holding individual vertex attributes, i.e. vertex position, normal, color, texture coordinates, et cetera, and additional user-defined constants. Likewise, a set of output registers can be written by the program that are in turn interpolated in the rasterization stage yielding fragment attributes. A limited amount of general purpose temporary registers is provided for arbitrary use by the programmer. However, more advanced programming concepts such as branching or looping were not supported by the first programmable vertex processors.

Programmable vertex processing was closely followed by the introduction of similar fragment processing capabilities, replacing the previously described configurable fragment processing units. Basically the programming model for these new programmable fragment processing units resembles the previously described vertex programs. The main difference is in the available input and output registers and the availability of additional instructions, such as texture sampling instructions.

Besides limited flow control capabilities there were a number of additional restrictions that made programming of the first fully programmable units in the graphics pipeline often a complicated task. Only a limited number of instruction slots and registers were available, considerably restricting the size of programs. Further restrictions affect, for example, the number of texture indirections, i.e. texture sampling operations depending on previously computed or sampled values, or missing bit-wise integer operations.

However, most restrictions have been alleviated step by step with the release of ever more powerful hardware in the last years. The transition to a complete floating point pipeline and the introduction of dynamic flow control, i.e. branches and loops, as well as texture lookups in vertex programs has added enormously to the flexibility of programmable graphics processors and allowed for a much wider range of algorithms to be implemented on the GPU. Most algorithms presented in this thesis would not be feasible without these new possibilities.

Nowadays, modern GPUs, beginning with NVIDIA's GeForce 8 series and AMD's Radeon HD 2xxx series, feature a unified shader architecture that supports the same instruction set and features for both the vertex shader and the fragment shader, save for a few exceptions like the fragment kill instruction. Instead of the

distinct hardware units for vertex and fragment processing characterizing previous GPU generations, these new GPU architectures feature a set of identical processing units that are allocated to a specific processing task using a thread scheduler. Furthermore, these core processing units more and more resemble traditional CPU cores and support single IEEE floating point precision, efficient flow control, and real integer arithmetic.

High-Level Shading Languages Accordingly, the massive progress in hardware technology was accompanied by a corresponding evolution of the programming APIs. The next step towards general programmability of graphics hardware was heralded by the introduction of suitable high-level shading languages, such as the DirectX *High Level Shading Language* (HLSL) by Microsoft, NVIDIA's *C for Graphics* (Cg), and the ARB standardized *OpenGL Shading Language* (GLSL) that replaced the previously common assembly languages. Both GLSL [175] and Cg [139, 51] are adaptations of the C programming language to the specific needs of GPU programming. That is, they provide a number of language features that are specific to programming graphics hardware. For example, in addition to the elementary types they provide also basic types and operations for vectors and matrices commonly required in computer graphics and a set of equally accommodated standard library functions. Some other C concepts, such as pointers, have been eliminated, however, in order to simplify the language and to accommodate for the simplicity of the graphics hardware's processing units.

The availability of high-level programming languages not only made the development of shader code much simpler, easier, and less error prone, but also added a new level of hardware abstraction and, therefore, of shader portability. In contrast to assembler programs, which in general have to be hand-optimized for different hardware architectures, a high-level shader compiler can provide higher level program optimizations and on-the-fly optimizations for the underlying GPU architecture. Furthermore, it provides the possibility of transparent management of shader resources, for example temporary register allocation. In many cases the code generated by an optimizing shader compiler will provide higher performance than a hand-optimized solution since modern GPUs exhibit characteristics that in their complexity and mutual dependency are hardly comprehensible to the programmer anymore. Typical examples include automatic vectorization with respect to the SIMD instruction set of the processing unit or the exploitation of potentially available co-issue capabilities of the vertex or fragment processors, i.e. for example the parallel execution of a vector and a scalar operation. Furthermore, each GPU has its own specifics and particularities often only known to the hardware manufacturer. Therefore, programs written on the assembly level may not provide optimal performance.

Programmable Primitive Processing The most recent major change to the rendering pipeline was the introduction of programmable primitive processing. The newly introduced *geometry shader* extends parts of the standard pipeline's primitive assembly stage. Instead of immediately clipping and projecting the primitives setup in the primitive processing stage following vertex processing and primitive assembly they are treated as input to a geometry shader program. Conceptual, it works on a per–input-primitive basis and computes a theoretically arbitrary number of new primitives that may also differ in type from the input primitive. Thus, although, input to this stage is only a single primitive and its accompanying vertex attributes, a variable amount of output primitives and accompanying vertices can be generated; possibly discarding whole input primitives. While, both vertex and fragment shaders are still closely oriented to the stream processing model in that they retain the one-to-one relationship between input and output data, i.e. for each input vertex or fragment a single output vertex or fragment is generated respectively, in the geometry shader stage this concept is broken. This, however, for the first time provides a real possibility for generating geometry data on the GPU.

Furthermore, in the course of introducing the geometry shader, the possibility for direct streaming of vertex data to GPU buffer objects has been added. This so-called *transform-feedback* mode allows for capturing vertex data prior to clipping in the new primitive processing stage. The vertex attributes recorded are those emitted by the geometry shader stage if a geometry shader program is active. Otherwise, vertex attributes transformed by the vertex shader or fixed-function vertex processing are captured. This combination provides the capability for a large number of new algorithms, like multi-pass geometry processing, to be implemented using the GPU.

2.3 Programmable Rendering Pipeline

Figure 2.3 depicts the programmable rendering pipeline that can be found in contemporary GPUs. While there is still some fixed-function functionality left, most notably the rasterization, per-fragment test and blending units, the major parts of the pipeline, i.e. the vertex processing, primitive processing, and fragment processing stages, have been replaced by freely programmable shader units. In fact, a modern graphics processing unit featuring an unified shader core more or less resembles a multi-core processor with additional peripheral modules dedicated to graphics specific tasks, i.e. input assembly, rasterization, or per-fragment operations. Furthermore, the possibilities to directly render into a texture map and to redirect the stream of transformed vertex attributes to buffer objects allows to feed fragment and vertex output back into the pipeline. However, conceptually the basic idea of a rendering pipeline is still applicable.

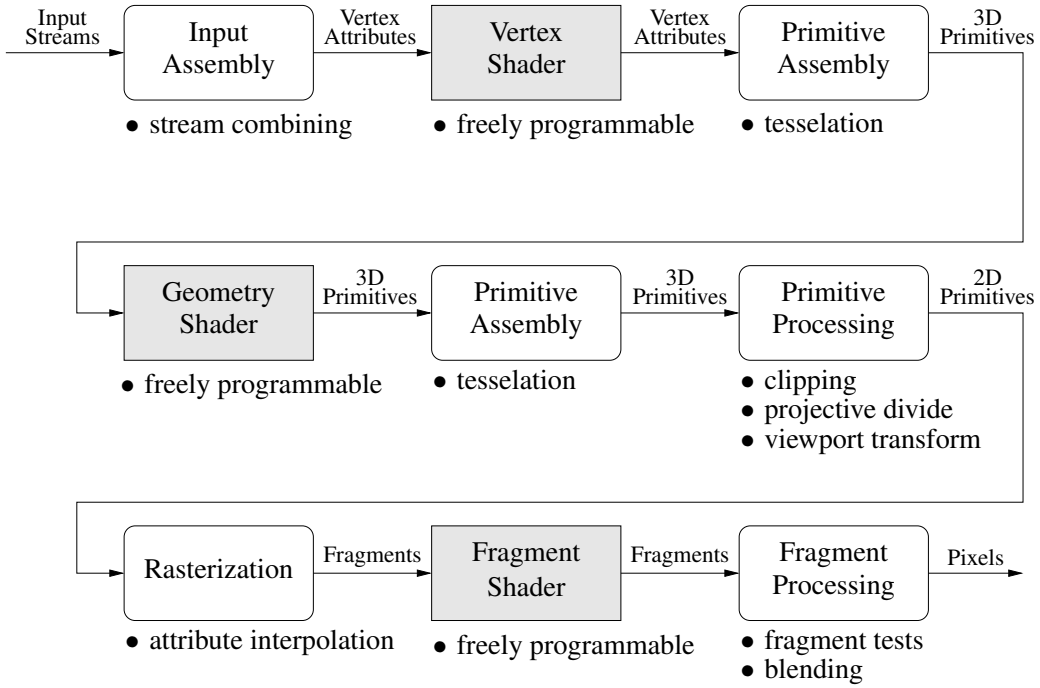


Figure 2.3: Simplified representation of the programmable rendering pipeline.

2.3.1 Vertex Processor

The programmable vertex processor replaces the fixed-function transform and lighting unit of former configurable GPU architectures. Its programmability is exposed by means of a vertex shader program that is executed independently for every element of the vertex attributes stream created in the input assembly stage. Vertex coordinates, lighting and colors, as well as texture coordinates can be freely processed within this program instead of being subject to the rigid predefined fixed-function geometry pipeline transformations. From a programmer's point of view input is provided by vertex *attribute* variables that vary per input vertex and are specified when drawing geometry and *uniform* variables that are constant for all invocations of the shader program. Recently also the capability to read from texture memory has been added to the input possibilities of the vertex shader. Output of the vertex shader are variables that are assumed to be *varying* linearly across primitives. In particular, these are the clip-space position of the vertex and any associated color and texture coordinate value. A vertex shader program can be either specified using low-level GPU assembly language instructions, like ARB assembly, or a high-level programming language, like GLSL.

2.3.2 Geometry Processor

The geometry processing stage in a sense redefines the primitive assembly stage of the fixed-function pipeline. Geometry shaders are executed after vertices have been processed in the vertex shader and have been assembled to primitives, but prior to primitive processing. As mentioned before, in a geometry shader program basic input primitives, such as a point, a line, or a triangle, are processed to generate an arbitrary number of new output primitives. That means instead of working only on a single vertex element, as in the vertex shader, it is possible to access topological information of the input geometry. In case of triangle primitives the input consists of the attributes of the three constituting vertices of each triangle. Furthermore, instead of emitting only a single output element, like a vertex, a stream of output primitives of predefined type is generated consisting of an arbitrary number of vertices.

Otherwise, a geometry shader program provides the same possibilities as the vertex shader, in other words, it works on the same kind of input data and essentially produces the same type of clip-space output data. This may cause the impression that the vertex shader has been made obsolete with the introduction of programmable geometry processing. However, for efficiency reasons separate vertex processing has still its legitimation. For example, a vertex of a triangle strip has to be processed only once in the vertex shader while in the geometry shader working on a per-primitive level the same vertex has to be processed up to three times independently.

2.3.3 Fragment Processor

Programmable fragment processing replaces the fixed-function (multi)texturing functionality. In particular, these are texture access, texture application, and final color sum computation. The fragment shader's inputs are the linearly interpolated vertex attributes after rasterizing the 2D primitives. Output, however, is restricted to one or more RGBA color values—depending on the number of available render buffers—and the depth value of the fragment. Most importantly, it is not possible to change the fragment's screen-space position within a fragment shader.

Another concept closely related to programmable fragment processing is the so-called early z-test. This test behaves essentially like the standard z-test applied as part of the per-fragment test with the only difference that it is applied before a fragment shader program is executed for a particular fragment. Thus, expensive per-fragment computations can be avoided by discarding fragments that are already occluded by pixels previously written to the framebuffer. The early-z test, however, is only effective in case the incoming fragment's depth value is not modified in the fragment shader and there is also no other possibility to prevent

fragments to be written to the framebuffer.

2.4 GPUs in Non-Graphics Applications

The exceptional parallel processing capabilities in combination with the steadily increasing amount of memory found on graphics cards, as well as the ease of programmability made available by high-level shading languages has initiated a trend towards using the GPU not only for graphics applications but also for general compute intensive applications. This has led to an entire new field of research: General-Purpose Computing on the GPU (GPGPU). Indeed, a great variety of grid-based computational problems, like fluid dynamics, image processing, or computer vision algorithms, exhibit strong data locality combined with a high-arithmetic density making them perfectly amenable to parallel processing on the GPU.

However, using graphics processors for non-graphics related purposes is in general not a new trend in the graphics community. Already the fixed-function pipeline inspired researchers to use its SIMD processing capabilities for non-graphics related applications. This includes, for example, work related to matrix multiplication [119], the computation of Voronoi diagrams [82], wavelet transformation [84], and nonlinear filters [85]. Yet, the availability of a complete floating point pipeline and the much improved programmability have led to a widening range of applications that use graphics hardware as a numerical co-processing unit. These include, for example, the solution of linear equations systems [111, 18] or the simulation of large astronomical n-body systems [163].

Although visualization algorithms are somewhat more directly related to traditional graphics problems than the aforementioned applications, in most cases they also exhibit the typical characteristics of a general stream computation problem. The visualization pipeline, as described in Chapter 3, typically operates on a stream of data elements transforming raw computational or measurement data into an image. While traditionally only the last step of this pipeline, the actual rendering of the image, was done on the GPU, the new possibilities allow to shift also major parts of the mapping and filtering stages to the GPU, harnessing not only the computational power of modern GPUs but also exploiting the much higher data locality offered by this approach. How this task can be accomplish in order to interactively visualize three-dimensional field data, will be the central theme of the remainder of this thesis.

In order to lift the burden of learning a graphics programming API from the application programmer a number of general abstract stream computation frameworks for GPUs have been proposed [143, 24, 120, 144]. Meanwhile, using the GPU as a parallel processor for general stream computing has drawn the attention

of hardware manufacturers and is now actively furthered by the remaining two major GPU companies by offering both specialized hardware and software support. Dedicated hardware is available in the form of NVIDIA Tesla and ATI/AMD FireStream boards. Both products basically feature the same GPU as employed in their high-end workstation product lines, i.e. NVIDIA Quadro and ATI/AMD FireGL respectively. However, parts that are unnecessary for compute applications, such as RAMDAC, output ports, et cetera, have been removed. In addition, they also provide specialized application programming interfaces. The *Compute Unified Device Architecture* (CUDA) [152] by NVIDIA provides an enhanced high-level programming language derived from the C programming language with some simple extensions realized by a pre-compiler. Functions are allowed to be scheduled for execution on both the host CPU as well as on the scalar stream processors of the G8x GPU architecture and its successors. Furthermore, low-level access to the hardware is provided by the *PTX* assembler interface [154]. Similar functionality is provided by ATI/AMD through the *Brook+* stream computing extension to C [2]—an extension to the BrookGPU language [24]—,which is build on a compute abstraction layer (*CAL*) for ATI/AMD GPUs that allows low-level programming of the GPU in a macro assembly language. Besides these vendor-specific solutions there are efforts under way to define standardized compute APIs for GPUs. There are currently two such projects: the *Open Compute Language* (*OpenCL*) [150]—a C extension similar to CUDA—and Microsoft's DirectX 11 *Compute Shader* [22].

On the other hand, the last few years have also seen the introduction of multicore CPUs, the Cell Broadband Engine architecture [61], and the revival of co-processor cards, like the Clearspeed simulation accelerator [28] boards, or the DRC reconfigurable FPGA-based processor units [43], which can be directly plugged into a free CPU socket. Although multicore CPUs are still in there infancy, while GPUs already feature a massively parallel processing core, it will be interesting to see how GPUs can compete against these and other specialized solutions in typical high-performance computing applications.

2.5 Future Prospects

In general it is very hard to predict what is coming next in the GPU market. Hardware manufacturers are usually quite reluctant to reveal information about future product generations. However, there is much dynamics in the graphics hardware industry. New architectures, such as Intel's *Larrabee* design [190], turn up and existing hardware, not least driven by new applications in the high performance computing market, keeps evolving at a fast pace. Therefore, it is foreseeable or at least likely that some of the things mentioned in the following will happen in the

near future.

Besides the obvious improvements such as higher clock frequencies, deeper pipelining, more shader units, larger texture memory and caches, wider and faster buses to texture memory as well as between the host CPU and the GPU, or even the integration of CPU and GPU on a single chip, there will be most likely a continued shift in GPU design towards the general stream computing paradigm. The core of future GPUs will most likely consist of a massive parallel array of freely programmable scalar processing units. Additional optional peripheral units, such as rasterizer or blending units, will provide the rest of the traditional graphics processing pipeline. Thus, it is easy to employ the same computational core as a traditional graphics processing unit or as a massively parallel stream processor.

So far, dedicated GPU hardware for stream computing has been basically a byproduct of the normal GPU development driven by the graphics and games industry. However, there are certain features essential for GPGPU applications that may change this. One is full 64-bit IEEE double precision floating point support. First GPUs that feature dedicated double precision processing units are already available. ATI/AMD's HD 38xx and 48xxx series as well as NVIDIA's G200 series provide native support for double precision computations. However, the number of double precision units and accordingly the available peak performance is only one fifth and one eights of single precision units and performance for the ATI/AMD GPUs and NVIDIA GPUs respectively. Furthermore, it can be expected that support for double precision computations will only be available through compute APIs like CUDA and CAL/Brook+ and will not be accessible by graphics programming APIs. Since a full double-precision rendering pipeline will be expensive due to the additional amount of transistor functions and in most cases is unnecessary for traditional graphics applications it is most likely that a complete switch to double precision could be only expected for specialized stream processing solutions or possibly in high-end graphics accelerator boards.

With the introduction of programmable primitive processing, i.e. the geometry shader, the last parts of the rendering pipeline, despite rasterization and input assembly, that remain fixed-function are the per-fragment tests and blending units. In many cases it would be desirable to have more fine-grained control over those per-fragment operations. Integrating those tasks into the fragment shader would require read-modify-write access to the framebuffer. This, however, has several consequences. First, it would lead to serious race conditions due to the parallel nature of fragment processing and therefore would require some kind of synchronization or locking concept. Second, it would require very large pixel data caches to hide the latency of accessing framebuffer memory. However, it is conceivable that we will see programmable blending units in future hardware generations.

Chapter 3

Fundamental Visualization Concepts

Scientific visualization is the computer-assisted transformation of abstract information, typically represented by numerical values, into a visual representation, e.g. still images, movies, or animations, in order to facilitate the understanding and analysis of the data. Although many other—possibly more precise and formal—definitions can be found in the literature, this short statement describes the gist of visualization. Visualization deals with abstract data that due to its complexity or its mere size is not directly accessible to human cognition and transforms it into a representation accessible to the human visual system. A prime example of everyday visualization is the television weather map, it conveys the information contained in several gigabytes of simulation data obtained from a complex numerical computer simulation using simple and intuitive metaphors easily intelligible to the layman.

This chapter serves the purpose of providing the necessary scientific visualization background required in the forthcoming chapters of this work. Basic visualization nomenclature is introduced. Utilizing the concept of the so-called visualization pipeline a brief overview of the general visualization process and the data structures involved is given with a special emphasis on the motivation for moving more and more parts of this pipeline onto the GPU. Finally, a basic classification of visualization techniques is provided.

3.1 The Visualization Pipeline

Similar to the concept of the rendering pipeline introduced in Chapter 2 the data transformations necessary to create a visualization, i.e. to generate the final image from the source data, is often depicted in the form of the so-called visualization

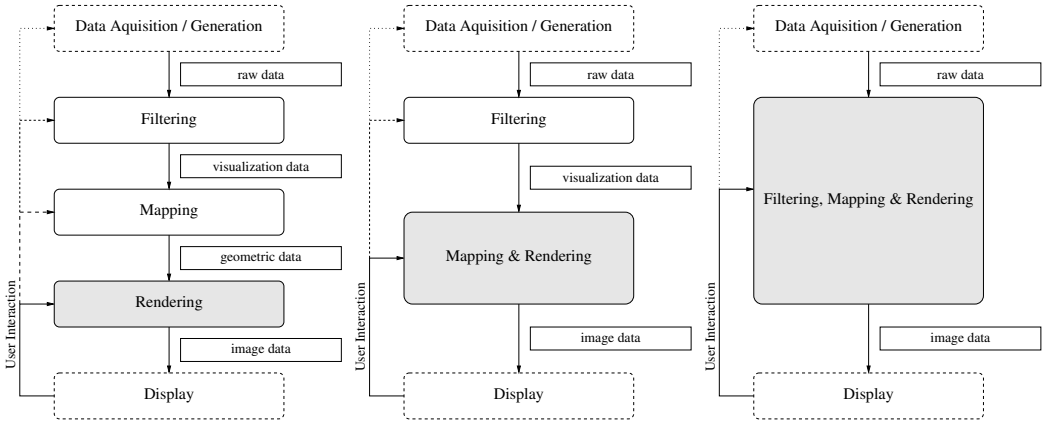


Figure 3.1: Main stages of the visualization pipeline. From left to right varying levels of GPU usage are shown: The classical visualization pipeline according to [70], GPU-based mapping and rendering, and a completely GPU-based visualization pipeline.

pipeline [70]. The different stages of the classical visualization pipeline, shown in the leftmost image of Figure 3.1, are data acquisition, data filtering, visualization mapping, rendering, and display of the visualization result. However, in many cases only filtering, mapping, and rendering are considered the main stages of the visualization pipeline, whereas data acquisition and display of the final rendering are considered preprocessing and postprocessing steps respectively. The data acquisition or data generation step involves the acquisition of the raw data fed into the visualization pipeline either by numerical simulation, e.g. CFD or FEM, or captured by sensor devices, e.g. CT or MR scanners.

Depending on the type of data or the acquisition process, the raw input data may require some kind of processing in order to improve the data quality or to remove unnecessary detail. This is the task of the subsequent filtering step. It performs operations such as data conversion, data reduction, or other more sophisticated data processing operations like denoising, segmentation, or resampling and interpolation. The resulting so-called visualization data is input to the central part of the visualization pipeline: the visualization mapping stage. In this step the information content contained in the visualization data has to be mapped to geometric data, i.e. graphical attributes like geometric primitives, color, or texture. This mapping of abstract data items to renderable graphics attributes corresponds to the actual visualization algorithm and constitutes the core of every visualization method. In the last step the geometric data is transformed into an image or a set of images by means of appropriate computer graphics techniques.

As is also shown in Figure 3.1, the visualization pipeline is not static. In fact it represents a user-controlled feedback loop, in which the user analyzing the data

may interact with the different stages of the pipeline.

In terms of performance and usability of a particular visualization tool the rendering step is the most critical part for interactive visualization. While the filtering and mapping in many cases can be precomputed, interactive manipulation of viewing parameters by the user as well as sufficiently high refresh rates are crucial for understanding, at least, 3D data. However, the level of interaction may vary depending on the particular visualization task. For example, only the ability of changing the mapping parameters on-the-fly allows for interactive exploration of unknown data. In the most extreme case it is possible for the user to influence the data generation process itself. This kind of interaction is commonly referred to as (computational) steering.

Although the model of the classical visualization pipeline is convenient for discussing basic visualization algorithms, it does not match many modern visualization techniques. First, and most important, many modern graphics-hardware–assisted visualization algorithms do not allow a clear distinction between the mapping and the rendering stage and therefore do not fit into the classic pipeline model. In pre-integrated direct volume rendering or any other post-shaded direct volume rendering algorithm, for example, both pipeline stages are combined into a single interpolation, classification, and rendering step. Second, in view of ever growing data set sizes it is often not sufficient to optimize a single step of the pipeline. Either, since it is not possible or efficient to store or to transfer the intermediate data or since the computations are not easily separated. Instead, for interactive exploration the efficiency of the complete pipeline has to be improved. One possibility is to harness the parallel processing power of modern programmable GPUs by consequent implementation and combination of as many pipeline stages as possible on the GPU.

The middle image and the right image of Figure 3.1 depict different degrees of GPU utilization. In all three images the gray boxes indicate parts of the pipeline that are realized using graphics hardware. In the first case only mapping and rendering have been combined, this corresponds, amongst others, to the aforementioned direct volume rendering techniques. Moving also the filtering part to the GPU provides the most advantages in terms of facilitating exploravity, since it allows the interactive manipulation of visualization parameters in all stages of the pipeline. The remainder of this thesis will address the problem of how to efficiently exploit the programmability and parallel processing capabilities of modern graphics processors for interactive visualization of three-dimensional data fields of varying data complexity and abstraction level.

3.2 Visualization Techniques

According to Post et al. [164] visualization techniques can be classified using a three element taxonomy: global techniques, geometric techniques, and feature-based techniques. Although their classification is specifically aimed at flow visualization techniques, it easily generalizes to other scientific visualization techniques as well. A global technique provides a qualitative, global visualization at a low abstraction level. Examples of this category are direct volume rendering algorithms or dense flow visualization techniques like Line Integral Convolution. Geometric techniques use geometric objects, i.e. curves, surfaces, solids, that are directly derived from the visualization data in order to visualize it. Prominent examples of this category are the Marching Cubes algorithm for isosurface extraction and streamline-based visualization of flow fields. Geometric techniques provide an intermediate-level of data abstraction as well as of locality. In terms of abstraction the highest level is provided by the extraction and visualization of characteristic features from the data field. Such features are often defined by mathematical or phenomenological description of a specific physical phenomenon, e.g. vortices or shock waves. The extracted features are either visualized using application specific glyphs or geometric techniques. Examples are visualizations based on scalar, vector, or tensor field topology or the extraction of vortex core lines.

Another, more common, distinction of visualization algorithms is to classify them according to the nature of the input data type and the dimensionality of the underlying domain [188, 207], i.e. algorithms for scalar, vector, or tensor fields.

The remainder of this thesis will be roughly structured according to those taxonomies. GPU-based techniques for global, geometric, and feature-based visualizations are discussed. Furthermore, scalar, vector, and tensor data fields are considered. In Chapter 4 a global GPU-based method for the visualization of three-dimensional scalar fields, i.e. volume raycasting, is presented while Chapter 5 describes the GPU-based extraction of polygonal isosurfaces; a geometric visualization technique. The next chapter, Chapter 6, presents a technique for point-based rendering of quadric glyphs that is employed for feature-based visualization of vector and tensor fields. This is followed by another feature-extraction technique in Chapter 7. Here a method for the topological analysis of vector fields is described.

3.3 Data Fields, Grids, and Interpolation

Although visualization data in general can be arbitrarily abstract, for example database entries, in this work we will restrict the discussion to data that is a discrete representation of three-dimensional data fields, i.e. data consisting of a finite

number of data points located at discrete spatial positions $\mathbf{x}_n \in D$ in a three-dimensional domain D. Each data point is represented by a tuple of data values of arbitrary numerical range and dimension. Depending on the dimensionality of the data tuple a distinction is drawn between scalar, vector, and tensor fields. Typical scalar quantities, i.e. 1-tuples, are temperature or pressure in a CFD data set. Accordingly, a three-dimensional flow field would be represented by vectors, i.e. 3-tuples, of real values while shear stress can be described as a second-order tensor field, i.e. 3×3 matrices. If not stated otherwise it is assumed that the data tuples $f_n := f(\mathbf{x}_n)$ describe a sampled representation of the underlying field $f(\mathbf{x})$ and that $f(\mathbf{x})$ is at least continuously differentiable.

A special type of data is termed multi-variate data, in this case multiple—possible uncorrelated—data tuples are given per data point. For example, a typical CFD data set consists of multiple scalar quantities—pressure, density, or temperature—and a vector-valued velocity field. In fact, most simulation results are multi-variate by nature. Furthermore, visualization data may represent time-dependent or nonstationary fields. This adds another dimension to the data domain. In this work, however, only stationary fields or single time steps of nonstationary fields will be considered.

In addition to the data type the structure of the data has to be defined. While the data type describes the dimensionality of a single data point the data structure defines the spatial organization of the data tuples in the domain D. This consists typically of a grid structure plus an reconstruction or interpolation scheme that defines how to reconstruct field values $f(\mathbf{x})$ at arbitrary positions $\mathbf{x} \in D$. The next two sections will provide an overview of typical grid types commonly encountered in scientific visualization and their respective interpolation schemes.

3.3.1 Classification of Grids

Basically, the structure of visualization data, as defined in the previous section, can be arbitrary. In its simplest form only spatial positions of the data points are specified. This gridless or scattered data does not explicitly define any kind of topological structuring of the data points. However, most often this structure is some kind of a grid that either implicitly or explicitly defines the connectivity between the data points. Furthermore, a grid not only defines the topological relationship of data points, but in most cases also implies an inherent continuity of the represented data. Every grid consists of nodes or vertices and cells formed by edges interconnecting the nodes and normally defines a partitioning of the data domain. Data values can be either specified cell-centered or node-centered. Whereas the latter case is the more common one since it allows intermediate values to be computed from the data samples, while the cell-centered approach permits only a single data value per grid cell, which leads to discontinuities at the cell faces.

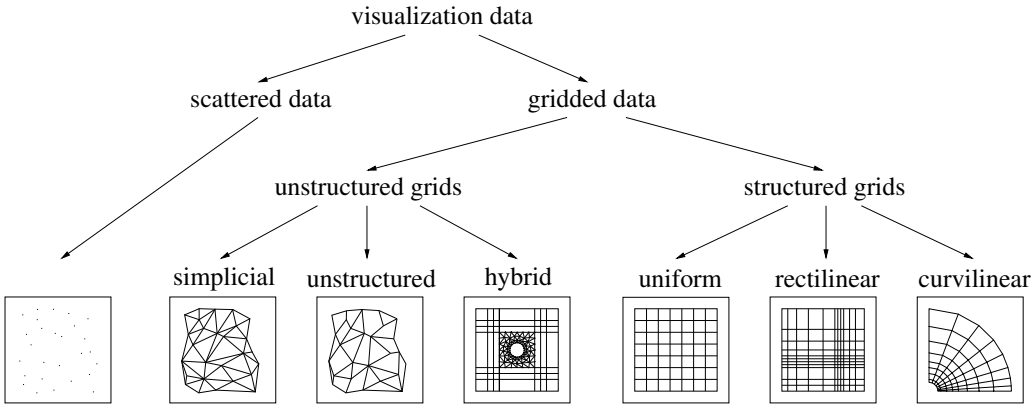


Figure 3.2: Classification of grid structures commonly encountered in scientific visualization.

In the following only node-centered grids are taken into account. Depending on the particular application domain or the employed simulation technique different grid types have been devised. Therefore, scientific visualization has to cope with a wide variety of different grids. The different grid types are commonly classified according to their topology and the shape of their grid cells. Figure 3.2 depicts a basic classification of grid structures commonly encountered in scientific visualization.

Basically, grids are distinguished into structured grids and unstructured grids. While structured grids are characterized by the fact that grid topology is implicitly defined, both the geometric positions of data points as well as their connectivity has to be stored explicitly in case of unstructured grids. This may be less efficient, both in terms of storage and data access, however, unstructured grids are very popular in FEM and CFD applications since they can easily adapt to complex geometries or boundary shapes. Since this work deals with both structured and unstructured grids as well as gridless data it is worth to take a closer look at their specific properties.

Gridless or scattered data Scattered data by its nature does not impose any kind of connectivity or topology on the data points. For each data point spatial coordinates have to be stored explicitly together with the actual data value. However, in many cases it is possible to infer connectivity in such a point cloud relying on interpolation to derive the assumed continuity. This might be the case if the data points represent samples of a otherwise continuous field. A typical approach in this situation would be to generate an unstructured grid based on a Voronoi tesselation. But there are also instances of scattered data where there is no implicitly defined grid structure or continuity and where it is not reasonable to impose these

properties on the data because of the strictly discrete nature of the data points. A typical source of such scattered data are numerical particle simulations.

Structured Grids Structured grids are the most common grid types in scientific visualization. Most numerical simulation algorithms and the majority of sensing devices, e.g. medical scanners, are working with structured grids. They are not only easy to handle programmatically but also provide efficient data access and data storage. There is no need to explicitly store connectivity information and in most cases even the node coordinates can be implicitly derived. From a given node with grid coordinates (i, j, k) its neighboring nodes can be directly computed by incrementing or decrementing the respective elements of the grid coordinates.

Structured grids are commonly subclassed into uniform grids, rectilinear grids, and curvilinear grids. In the simplest case, the cuboid cells of a uniform or regular grid form a orthogonal lattice with lattice parameters $(\Delta_x, \Delta_y, \Delta_z)$. The whole grid can thus be represented by a 3D array of data values. For each node geometric coordinates

$$\boldsymbol{x}^{\text{uni}}_{i,j,k} = \begin{pmatrix} i\Delta_x \\ j\Delta_y \\ k\Delta_z \end{pmatrix}, \quad i = 0, \ldots, N_x - 1;\ j = 0, \ldots, N_y - 1;\ k = 0, \ldots, N_z - 1,$$

can be directly derived from the node's grid coordinates. Here N_x, N_y, and N_z denote the resolutions of the grid for the respective coordinate axis. A special case of the regular grid is the Cartesian grid with $\Delta_x = \Delta_y = \Delta_z$. In respect of the graphics processing unit, textures maps are the direct analogon of uniform grids.

Rectilinear grids are more general than uniform grids in that the lattice parameters may vary with the grid index, i.e.

$$\boldsymbol{x}^{\text{rect}}_{i,j,k} = \begin{pmatrix} \sum_{l=0}^{i-1} \Delta_x[l] \\ \sum_{l=0}^{j-1} \Delta_y[l] \\ \sum_{l=0}^{k-1} \Delta_z[l] \end{pmatrix}, \quad i = 0, \ldots, N_x - 1;\ j = 0, \ldots, N_y - 1;\ k = 0, \ldots, N_z - 1.$$

Here, each $\Delta_d[.]$ with $d \in \{x, y, z\}$ denotes an array of $N_d - 1$ elements. Thus, the local resolution of a rectilinear grid can to a certain extent be adapted to the application's requirements.

In volume visualization, i.e. the visualization of 3D scalar data fields, cells of rectilinear grids are typically referred to as *voxels*—short for volume elements.

The most general structured grid is the so-called curvilinear grid. As in the case of the rectilinear grid the grid coordinates of a node can not directly mapped to the node's geometric coordinates. However, in contrast to the rectilinear grid,

it does not suffice to store a single 1D array per coordinate axis. Instead, the coordinate has to be stored either explicitly, i.e.

$$\mathbf{x}^{\text{curv}}_{i,j,k} = \begin{pmatrix} x[i,j,k] \\ y[i,j,k] \\ z[i,j,k] \end{pmatrix}, \quad i = 0,\ldots,N_x-1;\ j = 0,\ldots,N_y-1;\ k = 0,\ldots,N_z-1$$

or, alternatively, may be computed programmatically using coordinate functions $x(i,j,k)$, $y(i,j,k)$, and $z(i,j,k)$. In the common case of a uniformly-spaced polar grid these functions are given by

$$\begin{aligned} x(i,j,k) &= i\Delta_r \sin(\tfrac{2\pi j}{N_\Theta}) \cos(\tfrac{\pi k}{N_\phi - 1}), \\ y(i,j,k) &= i\Delta_r \sin(\tfrac{2\pi j}{N_\Theta}) \sin(\tfrac{\pi k}{N_\phi - 1}), \\ z(i,j,k) &= i\Delta_r \cos(\tfrac{2\pi j}{N_\Theta}), \end{aligned}$$

$$\text{with} \quad i = 0,\ldots,N_r - 1;\ j = 0,\ldots,N_\Theta - 1;\ k = 0,\ldots,N_phi - 1.$$

Where N_r gives the number of shells, N_Θ the number of latitudal segments, and N_ϕ the number of longitudal segments.

Unstructured Grids In contrast to structured grids, unstructured grids have no predefined geometry and topology, i.e. both vertex coordinates and vertex connectivity have to be stored explicitly. Although this increases the storage size and complicates data access, unstructured grids have a number of advantages that makes them attractive for simulation. They can easily adapt to complex bounding geometries and all naturally allow for locally adaptive grid resolutions.

The class of unstructured grids can be further subdivided in three subclasses: simplicial grids, general unstructured grids, and hybrid grids. The most popular unstructured grid is the simplicial grid composed of tetrahedral cells. The reason is simple; the tetrahedron is the only 3D cell shape that allows for linear data interpolation inside the cell, as will be discussed in the following section.

In a general unstructured grid there are no restrictions regarding the shape of the grid cells. Cells commonly found in such grids are tetrahedra, hexahedra, prisms, or pyramids. However, for the purpose of visualization such cells are again commonly decomposed into simplicial cells.

Obviously, all of the above mentioned grid types can be combined into a single unstructured grid. However, in case of a combination of structured and unstructured grids it can be beneficial to handle and store the respective parts of this combined grid separately instead of treating it as a general unstructured grid. Such a combination is called a hybrid grid.

3.3.2 Data Interpolation

Interpolation is the task of reconstructing intermediate values from the vertices' values for an arbitrary sampling point within a grid cell. Depending on the grid's type and the required continuity of the interpolant different interpolation schemes are employed. In the following only local interpolation schemes will be discussed, i.e. interpolation techniques that require only a relatively small number of the original data points in the vicinity of the interpolation position to find an interpolated data value. Global interpolation schemes, such as spline interpolation, in contrast, require expensive preprocessing and typically increase the data size. Already a 1D piece-wise cubic spline interpolant requires $4(n-1)$ coefficients to be stored for n interpolated data points.

Nearest-Neighbor Interpolation The simplest form of interpolation, nearest-neighbor interpolation, simply chooses the value of the data point nearest to the interpolation point. This yields a piece-wise constant reconstruction of the field and corresponds to assigning a constant data value to the Voronoi cell of each data point. The resulting reconstruction, however, is discontinuous at the faces of the Voronoi cells. In general this approach is very fast and easy to implement, but finding the nearest grid vertex might be quite involved in the case of a general unstructured grid. Since the basic concept of a nearest point does not require explicit connectivity information, nearest-neighbor interpolation also works for scattered data. However, there are other more sophisticated interpolations schemes that are specifically devised for gridless data, for example Inverse Distance Weighting [191] or interpolants based on Radial Basis Functions.

Nearest-neighbor interpolation for uniform grids is supported by the texture interpolation hardware of the graphics processing unit.

Linear Interpolation Linear interpolation is only possible in simplicial cells, i.e. lines in 1D, triangles in 2D, and tetrahedra in 3D. It is possible to describe any point $\mathbf{x}$ inside a simplex by an affine combination of the vertices $\mathbf{x}_n$ of the simplex. In case of a tetrahedral cell this is

$$\mathbf{x} = \alpha\mathbf{x}_0 + \beta\mathbf{x}_1 + \gamma\mathbf{x}_2 + \delta\mathbf{x}_3, \quad \text{with } \alpha+\beta+\gamma+\delta = 1.$$

Solving for the so-called barycentric coordinates $(\alpha, \beta, \gamma, \delta)$ of $\mathbf{x}$ with respect to the tetrahedron yields

$$\alpha = \frac{|\mathbf{x}, \mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}_2 - \mathbf{x}_3|}{|\mathbf{x}_0 - \mathbf{x}_3, \mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}_2 - \mathbf{x}_3|}, \qquad \beta = \frac{|\mathbf{x}_0 - \mathbf{x}_3, \mathbf{x}, \mathbf{x}_2 - \mathbf{x}_3|}{|\mathbf{x}_0 - \mathbf{x}_3, \mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}_2 - \mathbf{x}_3|},$$
$$\gamma = \frac{|\mathbf{x}_0 - \mathbf{x}_3, \mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}|}{|\mathbf{x}_0 - \mathbf{x}_3, \mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}_2 - \mathbf{x}_3|}, \qquad \delta = 1 - \alpha - \beta - \gamma.$$

Then, the interpolant is given by

$$\begin{aligned} f(\mathbf{x}) &= \alpha f_0 + \beta f_1 + \gamma f_2 + \delta f_3 \\ &= \alpha(f_0 - f_3) + \beta(f_1 - f_3) + \gamma(f_2 - f_3) + f_3. \end{aligned} \tag{3.1}$$

Trilinear Interpolation As mentioned before, linear interpolation as described above is only possible in simplices. However, it is possible to extend the 1D linear interpolation to higher dimensions by means of a tensor product approach. For a three-dimensional uniform grid this leads to the so-called trilinear interpolation scheme. Considering the 1D affine combination

$$x = \alpha x_0 + \beta x_1, \quad \alpha + \beta = 1$$

the formula for 1D linear interpolation can be similar to the linear interpolation formula of a tetrahedral cell easily derived as

$$f(x) = \alpha f_0 + (1 - \alpha) f_1, \tag{3.2}$$

with

$$\alpha = \frac{x_1 - x}{x_1 - x_0}.$$

Repeated application of (3.2) leads to the trilinear interpolation function for cell (i,j,k) of a rectilinear grid:

$$\begin{aligned} f(\mathbf{x}) = (1-\gamma)((1-\beta)((1-\alpha)f_{i+1,j+1,k+1} + \alpha f_{i,j+1,k+1}) + \\ \beta((1-\alpha)f_{i+1,j,k+1} + \alpha f_{i,j,k+1})) + \\ \gamma((1-\beta)((1-\alpha)f_{i+1,j+1,k} + \alpha f_{i,j+1,k}) + \\ \beta((1-\alpha)f_{i+1,j,k} + \alpha f_{i,j,k})), \end{aligned}$$

with $\mathbf{x} = (x, y, z)$ and

$$\alpha = \frac{\mathbf{x}^{\text{rect}}_{i+1,j,k} - x}{\mathbf{x}^{\text{rect}}_{i+1,j,k} - \mathbf{x}^{\text{rect}}_{i,j,k}}, \quad \beta = \frac{\mathbf{x}^{\text{rect}}_{i,j+1,k} - y}{\mathbf{x}^{\text{rect}}_{i,j+1,k} - \mathbf{x}^{\text{rect}}_{i,j,k}}, \quad \gamma = \frac{\mathbf{x}^{\text{rect}}_{i,j,k+1} - z}{\mathbf{x}^{\text{rect}}_{i,j,k+1} - \mathbf{x}^{\text{rect}}_{i,j,k}}.$$

Trilinear interpolation is the most widely used interpolation scheme for three-dimensional uniform and rectilinear grids. Its is comparatively inexpensive to compute, continuous at cell faces, and, most importantly for visualization applications, it is supported by graphics processing units. Its two-dimensional equivalent is called bilinear interpolation. Note, that trilinear interpolation is not a cubic interpolant. Instead, it varies linear along each straight line that is parallel to an edge of the cell, bilinear on each plane that is parallel to a face of the cell, quadratic on lines parallel to cell faces, and cubic for all lines that are not contained in any plane perpendicular to a principal axis of the cell.

Higher-Order Interpolation In visualization every interpolation scheme beyond the simple linear, bilinear, and trilinear schemes is considered to be of higher order. More complex schemes are typically considered too expensive to be evaluated in a visualization algorithm, especially if interactivity is a requirement. A comparatively inexpensive compromise is to use triquadratic or tricubic interpolation. Analogous to bilinear and trilinear interpolation triquadratic and tricubic interpolation can be be separated into a sequence of 1D quadratic and cubic interpolations respectively.

Currently, only nearest-neighbor, 1D linear, bilinear, and trilinear interpolation is directly supported by texture lookups on GPUs. However, there are techniques to efficiently compute higher-order interpolants by composing them of linear texture lookups [193, 109]. Using the techniques described by Sigg and Hadwiger [193] a tricubic interpolation can be achieved by only eight trilinear texture lookups in the data texture plus three additional linear lookups in an 1D texture of interpolation weights.

3.3.3 Estimation of Derivatives

Besides the reconstruction of a continuous representation of the sampled field another important problem is the approximate computation of derivatives of this field. For example, the gradient of a scalar field may be required as a normal in shading an isosurface. Since a rigorous discussion of derivative estimation on general grids is beyond the scope of this thesis, only the important cases of rectilinear and tetrahedral meshes will be discussed here.

Derivatives on Rectilinear Grids Although the trilinear interpolant is continuous at cell boundaries, it is not continuous differentiable over cell boundaries. Therefore, a common technique to compute derivatives on rectilinear grids is to compute vertex-centered approximate derivatives using *finite difference* approximations and to use interpolation to determine a value for intermediate points.

Most often a simple second-order *central differences* approximation

$$f'(x_i) \approx \frac{\Delta_x^2[i-1]f_{i+1} + (\Delta_x^2[i] - \Delta_x^2[i-1])f_i - \Delta_x^2[i]f_{i-1}}{\Delta_x[i-1]\Delta_x[i](\Delta_x[i-1] + \Delta_x[i])} \tag{3.3}$$

combined with first-order *forward and backward differences*,

$$f'(x_i) \approx \frac{f_{i+1} - f_i}{\Delta_x[i]} \quad \text{and} \quad f'(x_i) \approx \frac{f_i - f_{i-1}}{\Delta_x[i-1]}, \tag{3.4}$$

at grid boundaries are used due to their simplicity and computational efficiency. In addition to these standard low-order approximations it is also possible to derive finite-difference approximations with asymptotic errors of arbitrary order by

including a larger number of data points using derivatives of higher-order interpolation functions [54]. On the other hand, using a larger interpolation stencil further complicates the computation at grid boundaries.

In case of volume visualization, other popular approximation techniques for the gradient of discrete sampled volumetric data are discrete gradient filters based on edge filters known from image processing. Typical examples for such gradient operators employed in volume visualization are the $3\times3\times3$ Sobel operator or the $5\times5\times5$ Prewitt filter which combine low-pass filtering with gradient estimation to improve the quality of the approximation.

Derivatives on Tetrahedral Grids Since interpolation in tetrahedral cells is linear, the first order derivative is a constant for the whole cell. A cell-centered derivative can be easily computed from Equation (3.1). In case of a scalar data field its gradient is given by

$$\begin{aligned}\nabla \mathrm{f} =&(\mathbf{x}_1-\mathbf{x}_3)\times(\mathbf{x}_1-\mathbf{x}_3)\ (\mathrm{f}_0-\mathrm{f}_3)+\\ &(\mathbf{x}_0-\mathbf{x}_3)\times(\mathbf{x}_2-\mathbf{x}_3)\ (\mathrm{f}_1-\mathrm{f}_3)+\\ &(\mathbf{x}_0-\mathbf{x}_3)\times(\mathbf{x}_1-\mathbf{x}_3)\ (\mathrm{f}_2-\mathrm{f}_3).\end{aligned} \tag{3.5}$$

Obviously, this derivative is also not continuous at cell boundaries. A linear approximation of the gradient field, however, is commonly achieved by computing vertex-centered derivatives as a weighted sum of the constant derivative of adjacent grid cells using the tetrahedra volumes as weights.

Chapter 4

GPU-based Volume Ray Casting

4.1 Volume Visualization

Volume visualization in general is concerned with the visualization of three-dimensional scalar fields—or volumes—that arise in a multitude of application domains. The need for the visualization of volumetric data ranges from scientific visualization of simulations and measurements to the realistic rendering of gaseous and amorphous natural phenomena—fire, smoke, clouds—for special effects and games. Medical imaging methods, such as computed tomography (CT) and magnetic resonance imaging (MRI) are by far the most prominent source of volume visualization data. On the other hand, these methods today are also commonly utilized in nondestructive testing of materials. There is, however, a large number of other application domains that also rely on the analysis and interpretation of volumetric scalar data and require the visualization of data from sensor acquisition as well as numerical simulation, such as seismic measurement data, data resulting from computation fluid dynamics (CFD) simulations, et cetera.

In the following a 3D scalar field will be denoted by the map

$$s : \mathbb{R}^3 \to \mathbb{R}, \quad \mathbf{x} \mapsto s(\mathbf{x}). \tag{4.1}$$

Furthermore, we will assume, as long as not noted otherwise, that s is given as a sampled representation on a rectilinear grid and can be reconstructed by the trilinear interpolation scheme described in Section 3.3.2.

Providing an exhaustive overview of volume rendering techniques and their applications is beyond the scope of this work. Here, only a brief overview of the most common techniques and an introduction of the basic concepts necessary for understanding the following discussion will be provided. For an in-depth overview of modern interactive volume graphics the reader is referred to the excellent book by Engel et al. [49] that covers basic and advanced methods that have

been developed in the field of GPU-based volume rendering in the last couple of years.

Conceptually volume visualization techniques are differentiated in *indirect* and *direct* methods that differ in whether first an intermediate representation or feature is extracted from the data or an image is generated directly from the volume.

4.1.1 Indirect Volume Visualization

Indirect volume visualization methods generate an intermediate representation from the volume data. The most prominent example of indirect volume rendering is the reconstruction of isosurfaces, i.e. surfaces $s(\mathbf{x}) = c$ of constant scalar value embedded in the volume. The widely known *Marching Cubes* algorithm [132] is an example of a geometric visualization technique. It approximates an isosurface of the volume by a triangle mesh that can be efficiently rendered using standard graphics hardware. This approach fits very well to the classical visualization pipeline of distinct mapping and rendering steps.

An approach for the GPU-based parallel extraction of polygonal isosurfaces will be discussed in Chapter 5. In this chapter, however, we will focus on a direct volume visualization technique.

4.1.2 Direct Volume Visualization

The inherent reduction on a 2D surface and the unavoidable loss of information and context is the major drawback of indirect volume visualization methods. In contrast to indirect volume rendering approaches, direct volume rendering techniques aim for a direct, semi-transparent visualization of the scalar data set that avoids the conversion to an intermediate geometric representation and suits the in many cases fuzzy nature of 3D volumes much better than, for example, an isosurface. Furthermore, it is a global visualization technique that captures the entire 3D information contained in a volumetric data set in a single image.

The basis of direct volume rendering is a simplified model of light transport in participating media. Neglecting scattering and wavelength-dependent effects the transport of light along a ray through a idealized gas that emits and absorbs light can be described by means of the so-called low-albedo *volume rendering integral* [181, 228]. Most commonly this integral is written in the form [140]

$$\mathbf{I} = \mathbf{I}_0 e^{-\int_{T_0}^{T_1} \tau(t')dt'} + \int_{T_0}^{T_1} \tilde{\mathbf{c}}(t) e^{-\int_{t}^{T_1} \tau(t')dt'} dt. \tag{4.2}$$

Here $\mathbf{I}$ represents the light that reaches the camera along a ray $\mathbf{x}(t)$ that intersects the volume for the ray parameter interval $[T_0; T_1]$. The mappings

$$\tilde{\tau}(t) = \tau(s(\mathbf{x}(t)))$$

and

$$\tilde{\mathbf{c}}(t) = \tau(s(\mathbf{x}(t)))\mathbf{c}(s(\mathbf{x}(t)))$$

describe absorption and emission of light or, more generally speaking, map scalar values respectively, volume density s to optical properties, i.e. transparency and color. Finally, $\mathbf{I}_0$ is the light entering the volume at $\mathbf{x}(T_0)$ from the background of the scene.

The so-called *transfer functions* $\mathbf{c}(\cdot)$ and $\tau(\cdot)$ assign a vector of chromaticity coefficients and an opacity value to each scalar value based on an application-specific or user-defined classification. The appropriate specification of the transfer functions is the fundamental task in volume visualization. Only a suitable choice of transfer functions will give expressive visualizations and allows it to capture the features of interest while not obscuring them by unnecessary detail. Although in most applications one-dimensional transfer functions that depend only on the scalar value $s(\mathbf{x})$ are used, also so-called multi-dimensional transfer functions [105], taking additionally first and second derivatives of the scalar field into account, have been proposed.

Furthermore, often a local illumination term is added to the volume integral to emphasize material boundaries. Typically, it is assumed that the light reaching the local volume position is not obstructed or otherwise attenuated by other parts of the volume. This simple behavior can be modeled by modifying the source term of the emission absorption model using an additional source term derived from a local illumination model, like the Blinn-Phong model, which commonly depends on the actual ray position, the scalar value, and the scalar volume gradient at this position:

$$\tilde{\mathbf{c}}(t) = \tau(s(\mathbf{x}(t)))(\mathbf{c}(s(\mathbf{x}(t))) + \mathbf{c}_I(\mathbf{x}(t), s(\mathbf{x}(t)), \nabla s(\mathbf{x}(t)))). \tag{4.3}$$

Although analytical solutions of the volume rendering integral are possible for certain combinations of reconstruction filters and transfer functions, as has been shown by Williams and Max [228] for piecewise linear approximation of the scalar field along the ray and piecewise linear transfer functions $c(\cdot)$ and $\tau(\cdot)$, in the general case, however, it can be only approximated by numerical integration.

Dividing the integration interval $[T_0; T_1]$ in n equal-length segments of length $d = (T_1 - T_0)/n$ and splitting the second part of the volume rendering inte-

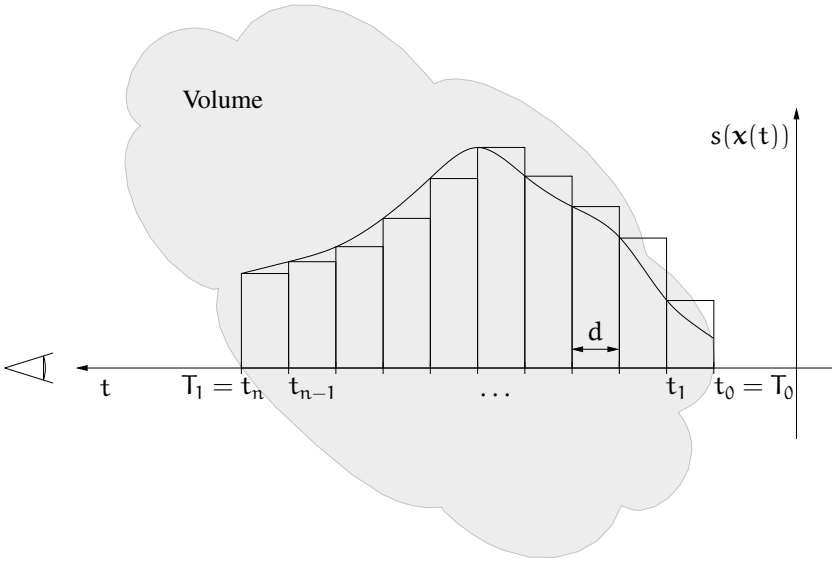


Figure 4.1: Discretization of the volume rendering integral by piecewise constant approximation of the scalar field.

gral (4.2) in n parts according to Figure 4.1 yields the following slab-based representation:

$$\begin{aligned}
\mathbf{I} &= \mathbf{I}_0 e^{-\int_{T_0}^{T_1} \tilde{\tau}(t')dt'} + \int_{T_0}^{T_1} \tilde{\mathbf{c}}(t) e^{-\int_{t}^{T_1} \tilde{\tau}(t')dt'} dt \\
&= \mathbf{I}_0 e^{-\sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} \tilde{\tau}(t')dt'} + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} \tilde{\mathbf{c}}(t) e^{-\int_{t}^{t_{i+1}} \tilde{\tau}(t')dt' - \sum_{j=i+1}^{n-1} \int_{t_j}^{t_{j+1}} \tilde{\tau}(t')dt'} dt \\
&= \mathbf{I}_0 \prod_{i=0}^{n-1} e^{-\int_{t_i}^{t_{i+1}} \tilde{\tau}(t')dt'} + \sum_{i=0}^{n-1} \prod_{j=i+1}^{n-1} e^{-\int_{t_j}^{t_{j+1}} \tilde{\tau}(t')dt'} \int_{t_i}^{t_{i+1}} \tilde{\mathbf{c}}(t) e^{-\int_{t}^{t_{i+1}} \tilde{\tau}(t')dt'} dt
\end{aligned} \tag{4.4}$$

Discretization of the remaining integrals in terms of Riemann sums, i.e. by employing a piecewise constant approximation of $s(\mathbf{x}(t))$ for each of the intervals, and neglecting self-attenuation when integrating over the i-th segment, i.e. assuming $\lim_{d\to 0} \int_t^{t_{i+1}} \tilde{\tau}(t')dt' = 0$, yields the discrete volume rendering equation

$$\mathbf{I} \approx \mathbf{I}_0 \prod_{i=0}^{n-1} e^{-\tilde{\tau}(t_i)d} + \sum_{i=0}^{n-1} \tilde{\mathbf{c}}(t_i) d \prod_{j=i+1}^{n-1} e^{-\tilde{\tau}(t_j)d}. \tag{4.5}$$

Instead of transparencies and absorption coefficients commonly opacities

$$\alpha_i = 1 - e^{-\int_{t_i}^{t_{i+1}} \tilde{\tau}(t)dt} \tag{4.6}$$

are used. Which, using similar arguments as have been employed in the derivation of Equation (4.5), can be further approximated by

$$\alpha_i \approx 1 - e^{-\tilde{\tau}(t_i)d} \approx \tilde{\tau}(t_i)d. \tag{4.7}$$

Thus, with $\mathbf{c}_i = \mathbf{c}(t_i)d$, Equation (4.5) can be rewritten to

$$\begin{aligned} \mathbf{I} &\approx \mathbf{I}_0 \prod_{j=0}^{n-1}(1-\alpha_j) + \sum_{i=0}^{n-1} \mathbf{c}_i\alpha_i \prod_{j=i+1}^{n-1}(1-\alpha_j) \\ &= c_{n-1}\alpha_{n-1} + (1-\alpha_{n-1})(c_{n-2}\alpha_{n-2} + \cdots \\ &\qquad \cdots (1-\alpha_1)(c_0\alpha_0 + (1-\alpha_0)\mathbf{I}_0)\cdots). \end{aligned} \tag{4.8}$$

This discrete approximation will converge to the correct solution of Equation (4.2) for $d \to 0$ or, in other words, for high sampling rates $1/d$.

Depending on the order the volume is traversed, two common recurrence relations are employed for evaluating Equation (4.8). First, *front-to-back compositing* samples the volume along the viewing ray from $\mathbf{x}(T_1)$, the point where the viewing ray enters the volume, to $\mathbf{x}(T_0)$, where the ray leaves the volume, which is described by

$$\begin{aligned} \mathbf{C}_{i-1} &= \mathbf{C}_i + (1-a_i)\alpha_{i-1}\mathbf{c}_{i-1}, \\ a_{i-1} &= a_i + (1-a_i)\alpha_{i-1}, \end{aligned} \tag{4.9}$$

with $\mathbf{C}_n = 0$, $a_n = 0$, and $I \approx \mathbf{C}_0 + (1-a_0)\mathbf{I}_0$. Consequently, *back-to-front compositing* evaluates the volume rendering integral by sampling the volume from $\mathbf{x}(T_0)$ on the backside of the volume to $\mathbf{x}(T_1)$. The iteration equation then becomes

$$\mathbf{C}_{i+1} = (1-\alpha_{i-1})\mathbf{C}_i + \mathbf{c}_{i-1}\alpha_{i-1}, \tag{4.10}$$

with $\mathbf{C}_0 = \mathbf{I}_0$ and $I \approx \mathbf{C}_n$.

Direct volume rendering implementations are typically divided according to the way the 3D volume is sampled. *Object-order* techniques, also called forward-mapping methods, process the volume cell-by-cell projecting the contributions of each cell onto the view plane. Depending on the order of traversal either front-to-back or back-to-front compositing is used to accumulate the color contributions. The three most prominent forward-mapping techniques are volume

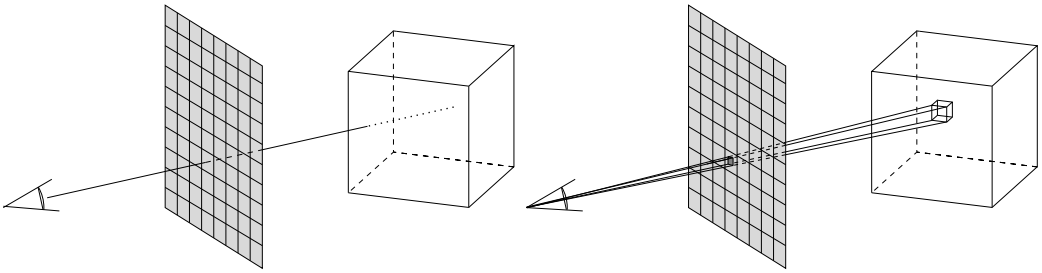


Figure 4.2: Image-order (left) and object-order (right) traversal of the volume.

splatting [224], cell projection [192], and slice-based hardware-assisted volume rendering [31, 25].

In contrast, *image-order* techniques, also known as *backward-mapping* techniques, iterate over all pixels of the image and evaluate the volume rendering integral for each pixel by traversing a ray through the volume in front-to-back order. In other words, color contributions are accumulated according to Equation (4.9). Ray casting is the most prominent image-order volume rendering algorithm.

The following discussion, will concentrate on the two most widely used volume rendering algorithms that are amenable to GPU-based implementations—namely slice-based volume rendering and volume ray casting.

Volume Ray Casting

Volume ray casting [93, 122, 123] is one of the first and, in fact, the most straightforward algorithm for evaluating the discrete approximation (4.8) of the volume rendering integral. The basic algorithm is relatively simple. It iterates over all pixels of the final output image. For each pixel one or more viewing rays originating from the viewing position are traced through the 3D volume. Colors and opacities are calculated at discrete sample points by reconstructing the scalar field at the samples using, e.g., trilinear interpolation, and subsequent application of the transfer functions. The resulting values are then accumulated using the front-to-back compositing formulae described previously.

Since the ray casting approach is the most natural and elementary implementation of direct volume rendering with respect to the evaluation of the volume rendering integral, it is often used as a reference in comparing the performance of volume rendering algorithms. On the other hand it features a number of additional advantages that made it the most popular volume rendering technique; it is easy to implement for both regular as well as unstructured grids [56], trivial to parallelize since each ray can be treated separately, and is amenable to various optimization techniques, such as early ray termination and adaptive sampling [123].

However, until recently ray casting was not easily implemented using graphics

hardware. Thus, only the use of special purpose hardware [145] or the utilization of massive-parallel computers [136] allowed interactive visualization. Ray casting has therefore been primarily used for high-quality off-line rendering on the CPU.

In Section 4.2.1 an approach for GPU-based ray casting will be discussed that exploits the inherent parallelism of modern programmable graphics hardware but also retains the advantages of easy implementation and flexibility.

Slice-based Direct Volume Rendering

Most of the work in interactive direct volume visualization in recent years has been focused on texture-based volume-slicing approaches. First introduced by Cullip and Neumann [31] and Wilson et al. [230], the evaluation of the volume rendering integral by sampling a 3D texture storing the volume data using a stack of viewport-aligned equidistant planar slices as proxy geometry has become the most widely employed volume rendering algorithm. The pixel-parallel processing during rasterization of the proxy geometry and texture mapping exploits the unmatched trilinear interpolation capability of modern graphics hardware and is the primary reason for the unsurpassed speed and success of this method. The actual volume integration is approximated by blending the textured slices in front-to-back or back-to-front order, according to the compositing formulae described previously, in the framebuffer.

Alternatively, also object-aligned slices can be used as proxy geometry for rendering a stack of 2D textures storing slices of the volume. However, depending on the viewing position the orientation of the slice planes has to be adapted as to minimize the angle between slice normal and viewing direction. This avoids excessive rendering artifacts when slices would be nearly parallel to the viewing direction. Accordingly, three different stacks of textures representing the same volume data are required. Figure 4.3 illustrates the differences between the two approaches.

The main drawback of the second approach is the additional memory that is required to store three times the same volume data. Furthermore, it requires additional effort to allow for arbitrary sampling distance and to avoid popping artifacts caused by switching the texture stacks. On the other hand, it is simple to implement, since it does not require the intersection of slice planes with the volume's bounding box, works with hardware that does not support 3D texturing, and offers high performance due to optimized 2D texturing hardware paths. However, nowadays using viewport-aligned slices combined with a single 3D texture is the prevalent graphics-hardware–assisted volume rendering technique.

Many enhancements to the basic approach have been proposed that exploit more advanced texture mapping capabilities of today's graphics hardware to increase the interactivity and applicability of the method. Some are closely related

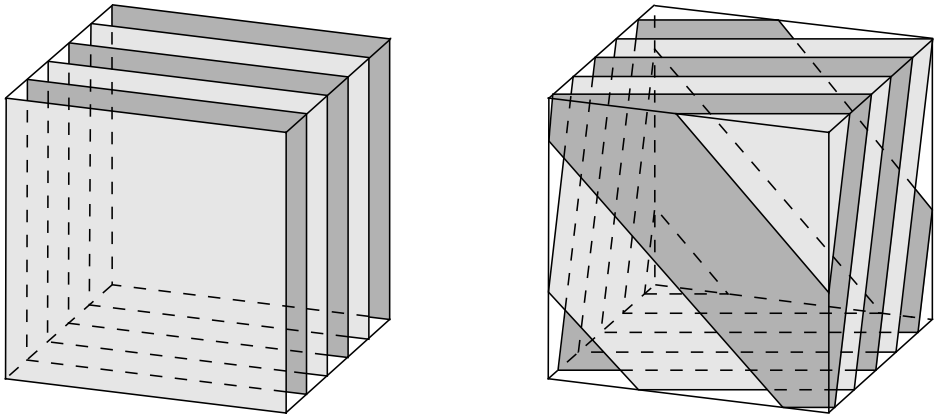


Figure 4.3: Slice-based direct volume rendering. Left: object-aligned slices with 2D textures. Right: viewport-aligned slices with 3D texture.

to the ideas presented in this chapter and will, thus, be discussed later on. For example, Westermann and Ertl [221] show how to render shaded isosurfaces using a slice-based volume renderer. Rezk-Salama et al. [172] employ the configurable fragment pipeline to achieve trilinear interpolation and arbitrary sampling distance in slice-based rendering using only 2D textures. Furthermore, they show how to efficiently render shaded isosurface and shaded volumes. The introduction of multidimensional transfer functions [105] and pre-integrated transfer functions [50] has significantly improved the quality of renderings that can be achieved. Also, some acceleration techniques proposed for the original ray casting approach, such as early ray termination and empty-space skipping [128], or hierarchical acceleration structures [17, 66, 118] have been successfully adopted to texture-based direct volume rendering. Nevertheless, even given the advanced programmability of modern GPUs, it is still much harder and requires considerably more effort to integrate such techniques into a slice-based volume renderer compared to the mostly straightforward implementation in an obviously much more flexible software-based ray casting code.

Nevertheless, until the advent of dynamic flow control support in programmable graphics processing units the prevalent technique for interactive visualization of volumetric scalar data has been slice-based direct volume rendering.

4.1.3 The Volume Rendering Pipeline

In Section 3.1 the concept of the visualization pipeline has been introduced. So, the questions is: How can direct volume rendering be mapped to the different stages of this pipeline? In the literature often an attempt is made to split the

different volume rendering algorithms into basic components and to describe the process of generating a volume rendering image by a volume rendering pipeline. Commonly four major computational components are identified: data traversal, i.e. the computation of sampling positions, interpolation, i.e. reconstruction of scalar values at the discrete sample points, classification, i.e. the application of the transfer functions, and compositing, i.e. accumulation of color contributions to the final rendering result. These four components can be directly mapped to the stages of the rendering pipeline. Sampling and interpolation corresponds to the filtering stage, classification maps scalar values to colors and opacity and therefore can be characterized as a mapping operation, and last, compositing of the color values and opacities corresponds to the final rendering of the image.

However, although in theory every volume rendering algorithm can be broken down to these basic elements and this model provides a conveniently abstract way of comparing volume rendering algorithms, it does not reflect the reality. As discussed in Section 3.1 direct volume rendering is a prime example for a visualization technique that does not match the rigid structure of the classical visualization pipeline. In theory it would be possible to implement a volume rendering algorithm that first computes all sampling positions, then interpolates data values for each of the samples, then applies the classification, and last combines the color values to the final image. Such an algorithm would be, however, impractical and inefficient. For all viable direct volume rendering methods, sampling, classification, mapping, and rendering can not easily be separated.

4.1.4 Volume Sampling

Although the discrete approximation of the volume integral given in Section 4.1.2 will converge to the correct solution for high sampling rates, i.e. $d \to 0$, the sampling rate that is necessary to avoid sampling artifacts might be too high for interactive rendering. As the sampling theorem states, a correct reconstruction of a signal is only possible with sampling rates higher than the Nyquist frequency. In volume rendering this is further complicated as not only the reconstruction of the original scalar field has to be taken into account but also the nonlinearity of the employed transfer functions. Thus, the sampling rate required to avoid undersampling the volume rendering integral is much higher than what would be necessary for sampling the volume data or the transfer functions separately.

Theoretical analyses of the appropriate sampling rate for direct volume rendering have been conducted by Kraus [107] and Bergner et al. [13]. Kraus suggests that the proper sampling frequency is proportional to the product of the Nyquist frequencies of the scalar field and the maximum of the Nyquist frequencies of the transfer functions. However, Bergner et al. note that this approach often is too conservative and overestimating in most practical situations. Instead they propose

a sampling criterion based on the product of the maximum Nyquist frequency of the transfer functions and the maximum gradient magnitude of the volume in the vicinity of the sampling location.

Given a suitable sampling criterion, there are two common solutions to the sampling problem. First, adaptive sampling can be used to adjust the sampling rate to the data. Although this approach has a long tradition in volume rendering, especially as a way to speed up the rendering process by adapting the local sampling rate to the lowest possible value for a given maximum acceptable error, a thorough theoretical analysis of the proper sampling distance taking also the effects of the transfer functions into account has been carried out only recently by Bergner et al. [13]. Practical applications of this approach will be discussed in Section 4.5.3.

Another solution that has been proposed is to separate the integration of the volume rendering integral from the sampling of the volume [142, 178]. Assuming a piecewise linear reconstruction $\bar{s}(l) = (1-l)s_f + ls_b$ of the scalar function between two consecutive sampling positions on the ray, the contributions

$$\begin{aligned}\alpha_i &= 1 - e^{-\int_{t_i}^{t_{i+1}} \tilde{\tau}(t)dt} \\ &\approx 1 - e^{-\int_0^1 \tau((i\bar{s}(l))dl}\end{aligned} \tag{4.11}$$

and

$$\begin{aligned}\mathbf{c}_i &= \int_{t_i}^{t_{i+1}} \tilde{\mathbf{c}}(t) e^{-\int_i^{t_{i+1}} \tilde{\tau}(t')dt'} dt \\ &\approx \int_0^1 \mathbf{c}(\bar{s}(l))\tau(\bar{s}(l)) e^{-\int_l^1 \tau(\bar{s}(l'))dl'} dl\end{aligned} \tag{4.12}$$

of this ray segment to the volume rendering integral depend solely on the scalar values corresponding to the sample points at the beginning and the end of the ray segment, $s_f = s(\mathbf{x}(t_{i+1}))$ and $s_b = s(\mathbf{x}(t_i))$, the distance d of those two sample points, and the transfer functions.

Thus, for fixed transfer functions the contribution of a single ray segment can be precomputed for a number of combinations of s_f, s_b, and d and stored in a 3D lookup table that can be indexed with respect to these three values. In a practical implementation this will be typically realized using a 3D texture.

The use of *pre-integrated transfer functions* provides a number of advantages. Since the computation of the lookup table can be done in a preprocessing step and

depends only on the transfer function, highly accurate integration schemes can be used for the computation, which are able to capture the behavior of high-frequency transfer functions. Thus, the problem of adequately sampling the volume rendering integral is reduced to sufficiently sampling the scalar field. Originally proposed by Max et al. [142] this concept has been first used by Röttger et al. [178] in the context of hardware-accelerated cell-projection for tetrahedral meshes and later used for hardware-assisted slice-based volume rendering by Engel et al. [50]. Due to the more accurate approximation of the scalar field and the accurate sampling of the transfer functions, this approach not only provides much higher quality for a given volume sampling rate but also allows larger sampling distances and accordingly higher performance while retaining the visual quality of the rendering result.

On the other hand, the additional memory requirements for storing the lookup table and the computational overhead for the pre-integration step, which has to be repeated whenever the transfer functions are changed, are the major drawbacks of this method. Therefore, several enhancements and acceleration techniques have been proposed. The main problem is the dimensionality of the lookup tables. The memory overhead can be drastically reduced in case uniform sampling distances can be assumed and, thus, a 2D lookup table would suffice. This is, in general, the case for ray casting and is also a reasonable approximation for equidistant viewport-aligned slices as long as no extreme perspectives are used [50]. A 2D table, however, is not sufficient for 2D object-aligned slices. In this case the use of logarithmically-scaled tables [108] might be beneficial.

Another problem is the computational overhead that in general prevents interactive updates of the transfer functions when using pre-integration. Several enhancements have been proposed that alleviate this issue, including the use of integral functions [50], the incremental computation of the integrals [218, 135], and parallel computation using graphics hardware [176].

Even when 2D tables are used and fast computation schemes are employed, the additional (texture) memory devoted to storing the pre-integrated color and opacity values is not least a limiting factor on the number of combinations of scalar values the pre-integration can be performed for. Therefore, another crucial factor that has to be considered when dealing with pre-integrated transfer functions is the choice of resolution of the pre-integration table. Here, a trade-off between accuracy, memory requirements, and computation time has to be made. In particular, this is a critical point for high dynamic range data, such as simulation results.

4.2 Ray Casting on the GPU

Although, as we have seen in the previous discussion, the use of graphics hardware has a longstanding tradition in interactive slice-based direct volume rendering, only the recent advances in programmability and the incredible pace at which the performance of graphics processors has increased over the last years, now provide the means of carrying the benefits of ray casting to hardware-accelerated volume rendering.

The reasons for investigating GPU-based ray casting techniques are many. The primary goal is often to achieve the same high-quality results as CPU-based ray casting implementations can provide but at interactive rates. This is accomplished by exploiting the possibility for straightforward realization of acceleration techniques, such as empty space skipping or early ray termination. Another reason why ray casting is so attractive is the simplicity and flexibility of the algorithm that allows the easy implementation of new or nonstandard optical models. And last, ray casting provides the means to overcome some of the shortcomings of the traditional volume-slicing approach, such as the time-consuming setup of the proxy geometry and the nonequidistant sampling due to the planar slice geometry.

Close investigation of volume rendering techniques reveals the fact that the dominating factor in most algorithms is the reconstruction of the scalar values for each sample point. The exploitation of the trilinear interpolation capabilities of graphics hardware is therefore the primary reason for the success of slice-based techniques for interactive volume visualization. So a long-term goal has been to combine the benefits of the same hardware acceleration with the advantages of the ray casting approach.

Several GPU-based implementations of the ray casting algorithm have been developed for both structured [110, 177, 198] as well as unstructured [218, 14] grids. According to the manner they achieve the sampling of the volume, these solutions can be further divided into two classes.

The first class [110, 177, 218, 14] performs multiple rendering passes—similar to traditional slice-based volume rendering—in order to advance the sampling position through the volume. Intermediate results, such as accumulated colors or the current sampling position, computed on a per-fragment basis are stored in temporary buffers that are accessed in subsequent rendering passes.

The second group executes the whole ray casting algorithm in a single rendering pass, exploiting the functionality of dynamic looping available on recent graphics processors. This allows the implementation of ray traversal, volume sampling, reconstruction, and integration in a single execution of a fragment shader program. Such an approach was first shown as a simple technology demo by NVIDIA [153] that demonstrates the use of advanced fragment shader functionality to implement a basic single-pass ray casting algorithm for volumetric in-game

effects. Stegmaier et al. [198] presented a framework for single-pass volume ray casting. This flexible framework is employed to demonstrate a number of non-standard volume rendering techniques, including translucent material and self-shadowing isosurfaces. Hadwiger et al. [71] presented a GPU-raycasting system for isosurface rendering. They employ a two-level hierarchical representation of the volume data set for efficient empty space skipping and use an adaptive sampling approach for iterative refinement of the isosurface intersections.

Since the first complete and published single-pass ray casting solution was developed as part of this thesis this approach will be discussed in more detail in the following. The devised framework for GPU-based ray casting was published in collaboration with Simon Stegmaier and Magnus Strengert [198].

Today, multi-pass ray casting techniques can be largely considered obsolete. Since the single-pass approach is much easier to implement and much more flexible, particularly with regard to optimization techniques, such as adaptive sampling or space skipping, the multi-pass approach is only interesting if older hardware has to be supported or for the study of certain implementation concepts that, for example, are used for efficient ray termination [110] and may be of interest for other applications as well.

4.2.1 Basic Single-Pass Ray Casting on the GPU

In contrast to a slice-based volume renderer, which requires laborious computation of the slice polygons, a basic single-pass GPU-raycaster is very straightforward and easy to implement. Basically it requires the rendering of a simple proxy geometry that creates the necessary fragments to trigger the execution of a fragment shader that realizes the ray-casting process for a single ray. In particular, the fragment shader performs ray setup, volume traversal, data interpolation and classification, and compositing. As for slice-based volume rendering with viewport-aligned slices, the volume data is stored in a 3D texture map.

The first part consists of rendering the front faces of the volume's bounding box defined by the pair of object-space points $(0,0,0)$ and (E_x, E_y, E_z) as proxy geometry and generates the necessary fragments that cover the perspective projection of the volume on the image plane. Here, the $E_c = \frac{(N_c-1)\Delta_c}{\max_l\{(N_l-1)\Delta_l\}}, c \in \{x,y,z\}$ denote the normalized object-space extents of the volume. For each fragment a viewing ray is set up passing through the respective image plane pixel. This is achieved by additionally providing the normalized object-space extents of the volume as texture coordinates, either directly or via a vertex program, to the vertices of the bounding box. Linear interpolation of these per-vertex attributes during rasterization provides each generated fragment with an object-space intersection point between the incident viewing ray and the volume.

```
// Input textures
uniform sampler3D VOLUME;
uniform sampler1D TRANSFERFUNCTION;

// Sample distance in object space
uniform float sampleDist;

// Transformation factors object space -> texture space
uniform vec3 scaleFactors;

void main(void)
{
    // The ray's intersection with the volume's front face
    vec3 pos.xyz = gl_TexCoord[0].xyz;

    // Compute ray direction from eye point position and ray start position
    vec3 dir =  sampleDist*normalize(pos - gl_ModelViewMatrixInverse[3].xyz);

    // Move to texture space
    dir *= scaleFactors;
    pos *= scaleFactors;

    gl_FragColor = vec4(0.0);

    do {
        // Sample volume at current ray position
        float scalar = texture3D(VOLUME, pos).r;

        // Look up color and opacity in tabulated transfer function
        vec4 src = texture1D(TRANSFERFUNCTION, scalar));

        // Perform front-to-back compositing
        gl_FragColor += (1.0 - gl_FragColor.a)*src;

        // Advance sample position
        pos += dir;

    // Test ray termination criterion
    } while (pos.x >= 0.0 && pos.x <= 1.0 && pos.y >= 0.0 && pos.y <= 1.0 &&
             pos.z >= 0.0 && pos.z <= 1.0);
}
```

Figure 4.4: Basic single-pass GPU-based volume ray casting code written in the OpenGL Shading Language.

Figure 4.4 shows the most basic fragment shader code that realizes a complete ray casting solution. First, the object-space intersection point on the volume's front face, which is the starting point of the ray traversal, is retrieved from the interpolated texture coordinate. Then, the direction of the viewing ray is computed by subtracting the start position from the camera or eye position, which is the image of the homogeneous view-space origin in object space and can, thus, be conveniently accessed as the fourth column of the inverse model-view matrix.

Since for convenient data access all following operations will be carried out in texture space, both ray origin and ray direction have to be transformed accord-

ingly, which is a simple nonuniform scaling operation using the constant scale factors $S_c = 1/E_c, c \in \{x, y, z\}$. In case of the ray direction it is important to transform the already normalized direction vector to ensure a uniform object-space sampling distance. This accounts for nonuniform sample distances as well as nonuniform dimensions of the volume.

In the actual ray traversal loop the volume data, stored in the 3D texture `VOLUME`, is sampled using the built-in trilinear interpolation capability of the graphics hardware for reconstructing the scalar field at the current ray sampling position. The thus retrieved scalar value is in turned used for looking up the color *RGB* and opacity α values in the tabulated transfer function stored in the 1D texture `TRANSFERFUNCTION`. Next, color and opacity are accumulated to the final fragment color according to the front-to-back compositing formula given in Section 4.1.2. Then, the current sample position is advanced along the ray according to the specified fixed object-space step size.

Finally, the ray traversal loop is terminated when the new sample position is outside the volume's extents. Since all operations are carried out in texture space, this condition is easy to check. Ray traversal is only continued as long as the three coordinates of the current sample position are greater or equal to zero and less or equal to one. Note that this simple test criterion applies only to recent graphics architectures that support non-power-of-two textures. If texture sizes supported by the target hardware are restricted to powers of two the upper boundary and the scale factors have to be adapted accordingly to account for the necessary padding of the volume data stored in the 3D texture. Furthermore, since older GPUs have a rather low limit on the maximum number of loop iterations that are supported (typically 255 iterations), it might be necessary to nest two traversal loops.

This is the simplest and most straightforward implementation of single-pass volume ray casting. In the remainder of this chapter several enhancements and optimizations, in terms of advanced volume shading, rendering performance, and image quality, will be discussed.

4.2.2 Generalized Ray Setup

The basic ray casting algorithm, as described in Section 4.2.1, is only suitable for viewing the volume data from the outside. Rendering only the front-facing polygons of the volume's bounding box, no fragments are generated for rays starting inside the volume since parts of the volume bounding box will be clipped by the view frustum. However, the possibility for viewing the volume from the inside, i.e. parts of the volume being clipped by the near clipping plane, is important for applications requiring volume fly-throughs, such as virtual endoscopy.

A correct ray setup for viewing positions inside the volume can be achieved by rendering not only the six faces of the volume bounding box as proxy geom-

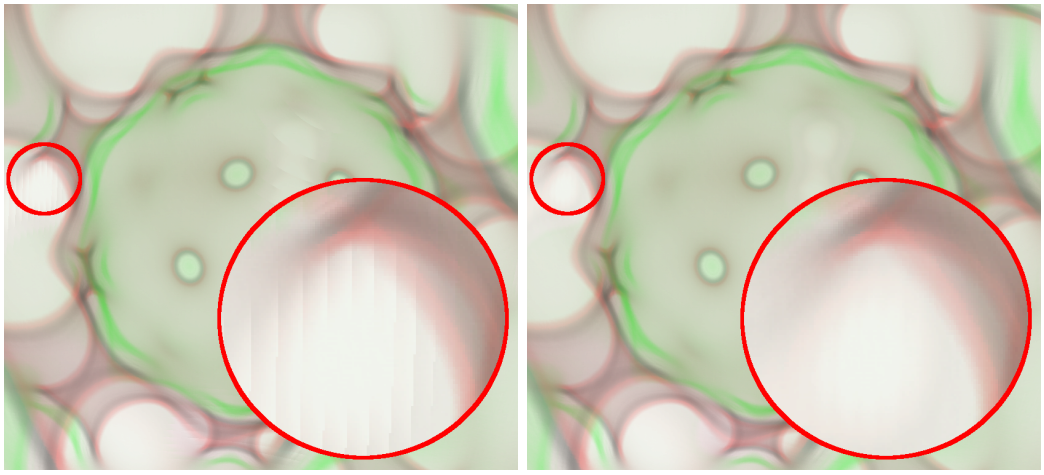


Figure 4.5: Comparison of GPU-based ray casting without (left) and with correct ray termination (right).

etry but also the intersection polygon of the bounding box and the near clipping plane. Using a fast method for computing the intersection polygon based on table lookups [200] these computations can be done very efficiently. Therefore, the overhead compared to rendering just the bounding box faces remains negligible.

Another, image-based solution, using the stencil buffer and multiple rendering passes to close holes resulting from near clipping plane intersections, has been proposed by Stegmaier [197]: First, clear the stencil buffer to zero and render the backfaces of the volume's bounding box in a stencil-only pass, incrementing the stencil buffer value for each created fragment. Then, enable the ray casting fragment shader and render the frontfaces of the volume; again, incrementing the stencil value for the respective fragments. Last, render the near clipping plane as proxy geometry restricting framebuffer updates to fragments where the stencil buffer value equals one. This will fill any remaining portion of the volume's bounding box projection not already covered in the previous pass. However, this solution depends on the availability of an early stencil test to be efficient.

4.2.3 Accurate Ray Termination

Besides correct ray setup and efficient termination of the integration process when the ray leaves the volume, accurate termination of the ray is another important point that has to be considered for high-quality volume rendering. The simple ray casting code shown in Figure 4.4 does not take care of the correct length of the last ray segment. Actually, the ray sampling is stopped too early, since the loop is simply terminated as soon as the next sample would be located outside the unit

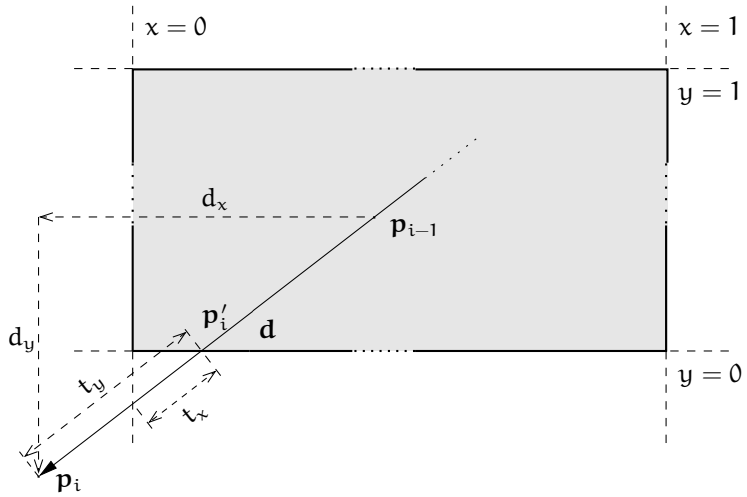


Figure 4.6: Computation of the correct ray exit point for accurate ray termination. Illustrated in the 2D case.

texture coordinate cube. In certain cases, especially for highly transparent transfer functions, this leads to noticeable artifacts as can be seen in Figure 4.5. Banding artifacts due to the sudden change in the number of sampling steps are clearly visible in the left image, which was rendered without correct ray termination.

Correct ray termination is also of particular importance when rendering large volumes that do not fit into texture memory and therefore have to be processed in multiple bricks [83] or in parallel volume rendering [149] to avoid artifacts at brick boundaries.

Finding the actual ray exit point $\mathbf{p}'_i$ is a special case of ray-box intersection. While several highly efficient algorithms for the general problem have been developed, our case is in fact much simpler and can be solved even more efficiently. Figure 4.6 illustrates the problem in the two-dimensional case. The first and most important difference to the general case is the fact that the existence of an intersection is always guaranteed. Second, the three out of six possible faces of the bounding box that are candidates for an intersection are determined by the signs of the components of the view direction vector $\mathbf{d}$, i.e. for $d_x < 0$ or $d_x > 0$ the face, which has to be checked, is $x = 0$ or $x = 1$ respectively. The same applies to the remaining directions. Computing the three possible ray parameter values is, again, rather simple since the texture-space bounding box of the volume is the unit cube, i.e.:

$$t_c = \frac{\left.\begin{array}{l} 0;\ d_c < 0 \\ 1;\ \text{else} \end{array}\right\} - p_c}{d_c},\ c \in \{x, y, z\} \tag{4.13}$$

Then, the smallest value t_{min} of the three computed negative ray parameter values corresponds to the sought-after intersection point that can be subsequently determined. Hence, it is sufficient to add the following few code lines at the end (after the loop) of the fragment shader shown in Figure 4.4 in order to correctly terminate the ray at the intersection with the bounding box.

```
// determine intersected faces
vec3 faces = vec3(greaterThan(dir, vec3(0.0)));

// compute ray-plane intersections
vec3 l = vec3(0.0);
if (dir.x != 0.0) l.x = (faces.x - pos.x)/dir.x;
if (dir.y != 0.0) l.y = (faces.y - pos.y)/dir.y;
if (dir.z != 0.0) l.z = (faces.z - pos.z)/dir.z;

// find closest intersection
float l_min = min(min(l.x, l.y), l.z);

// compute exit point
pos += l_min*dir;
```

Due to hardware support for register writemasking, the above shown high-level shader code translates to an efficient sequence of only ten GPU assembly instructions without any explicit branching.

Now, the volume texture can be sampled at the correct exit position and the corresponding color and opacity contribution looked up in the transfer function table. Since the resulting opacity α'_i and color contributions $\mathbf{c}'_i$ are defined per unit length for a given fixed sampling distance d, these values must be adapted to the new sampling distance d' before they can be combined with the previously accumulated contributions. According to Equation (4.7) the opacity values stored in the transfer function table are defined with respect to the original sampling distance d as

$$\alpha_d \approx 1 - e^{-\tilde{\tau} d}. \tag{4.14}$$

Thus, the corrected opacity values for the new sampling distance d' are given by

$$\begin{aligned} \alpha_{d'} &\approx 1 - e^{-\tilde{\tau} d'} \\ &= 1 - (1 - \alpha_d)^{\frac{d'}{d}}. \end{aligned} \tag{4.15}$$

Since the transformation from object space to texture space is an affine mapping, the sample spacing ratio d'/d can be directly computed as follows:

$$\frac{d'}{d} = \frac{|\mathbf{p}'_i - \mathbf{p}_{i-1}|}{|\mathbf{p}_i - \mathbf{p}_{i-1}|} = \frac{|\mathbf{d}| - |t_{min}\mathbf{d}|}{|\mathbf{d}|} = 1 + t_{min}. \tag{4.16}$$

Note that $\mathbf{d}$ is not a unit vector, as it has been scaled during the transformation to texture space and additionally has been multiplied by the sampling distance

d. Also note that $t_{min} \leq 0$. Sampling, classification, opacity correction, and compositing for the final ray segment is then achieved by the following additional shader instructions:

```
// Sample volume at ray exit position
float scalar = texture3D(VOLUME, pos).r;

// Look up color and opacity in transfer function
vec4 src =  texture1D(TRANSFERFUNCTION, scalar);

if (src.a != 0.0) {

    // Compute corrected opacity
    float newAlpha = 1.0 - pow(1.0 - src.a, 1.0+l_min);

    // Account for premultiplied alpha
    src *= newAlpha/src.a;

    // Perform blending
    gl_FragColor += (1.0 - gl_FragColor.a)*src;
}
```

Hence, with only approximately two dozen lines of high-level shader code a complete volume ray casting solution with accurate ray termination can be implemented. In Figure 4.6 the difference between accurate ray termination and the simple approach of Section 4.2.1 is demonstrated. Correct ray termination as illustrated in the right image successfully removes the disturbing wood-grain artifacts visible in the left image.

4.3 Generic Volume Visualization Framework

It is difficult to compare volume rendering results achieved with different volume rendering solutions. Especially, when each technique is implemented as a separate application. The attainable image quality and performance depends strongly on the underlying hardware, the used viewport size and the fraction of the viewport covered by the image of the volume, the current view direction, and other rendering parameters such as sampling step size, isovalue, or transfer function. In order to provide the opportunity for easy comparison of different volume rendering algorithms under the same conditions, a portable, flexible, and easily extensible volume rendering framework has been developed. Although the main objective is to provide a common framework for the rapid development and prototyping of GPU-based volume rendering solutions, the framework's structure is versatile enough to allow for the implementation of other volume rendering approaches or arbitrary visualization algorithms as well. For example, the point-based techniques for glyph-based visualization of tensor fields presented in Chapter 6 have been implemented in the same framework. In the following a brief overview of

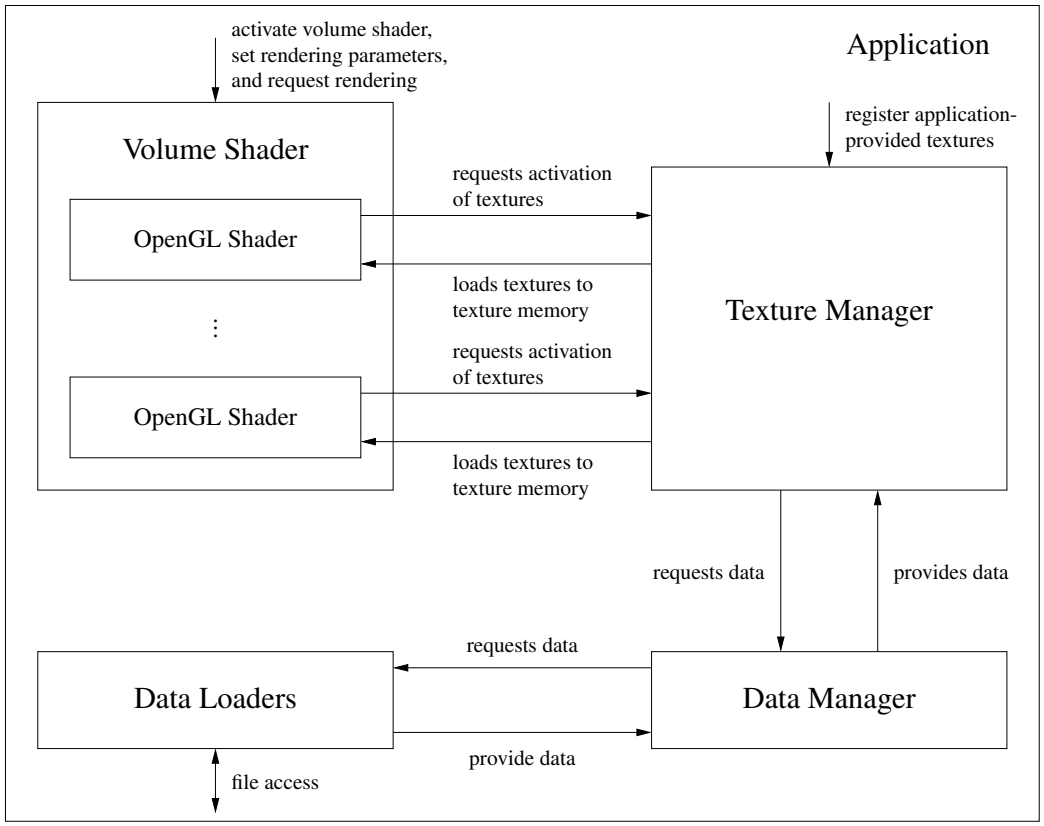


Figure 4.7: Overview of the main components of the volume shader framework.

the framework's architecture will be given and the concept of *volume shaders* will be introduced.

4.3.1 Overview and Architecture

The core of the volume visualization framework is realized as a shared library that allows its uncomplicated integration in different applications. This core consists of four major components. Figure 4.7 gives a brief overview of the main components and their interaction.

Volume shader objects are the central components of the system. The volume shader concept was introduced to encapsulate the actual volume rendering algorithm providing a common interface to the application. Volume shaders are realized as plug-ins by separate shared objects that can be dynamically loaded during runtime using the dynamic linking facility of the operating system.

The *texture manager* is responsible for the management of texture resources. This includes, for example, creation and deletion of textures on demand and the

refresh of time-dependent textures. These are either application provided textures, like a transfer function table, or dynamic textures that are created on request from a volume shader object. Examples for dynamically allocated and provided textures are textures storing the volumetric data, gradient information, or other derived or related fields.

The data for dynamic textures is in turn provided by the *data manager*. This module is responsible for managing and calling *data loader* plug-ins that eventually prepare the data, i.e. load it from the file system and perform preprocessing, such as the computation of a gradient field.

The application is only required to provide an OpenGL context, to activate the desired volume shader, to set up rendering parameters, such as view parameters, sampling distance, or isovalue, and to call the render routine of the volume shader. The volume shader then is responsible for the actual rendering, e.g. GPU-based ray casting, setting up OpenGL state, activating the required shader programs, and drawing the proxy geometry.

4.4 Optical Models and Rendering Modalities

The generality and flexibility of the ray casting approach allows for the simple realization of a large number of optical models that go beyond the already presented emission-absorption model, which has been applied in the basic direct volume rendering approach. Most of the optical models for direct volume rendering described by Max [140], including the most common emission-only, absorption-only, and local illumination models including shadows, require only slight modifications to the previously presented code.

In addition, however, there are many volume rendering modalities that have practical relevance but cannot be described by the volume rendering integral; for example *maximum intensity projection* (MIP), which is very popular in medical applications to achieve an X-ray like visual impression.

While volume rendering, as it has been discussed till now, is mostly concerned with scientific visualization of scalar fields, other applications, such as the photo-realistic rendering of semi-transparent objects, require a physically more accurate modeling of the light transport in a volume to achieve more sophisticated optical effects. In Section 4.4.3 a model for refracting volumes that includes selective chromatic absorption and simple scattering will be shown.

The following selection of volume rendering approaches will also exemplify the flexibility of the GPU-based volume ray casting approach and the framework presented above.

4.4.1 Pre-integrated Direct Volume Rendering

As has been discussed in Section 4.1.4, using pre-integrated transfer functions has a number of advantages that make them especially attractive for interactive high-quality volume rendering. Due to more accurate approximation of the scalar field and high-accuracy sampling of the transfer functions independent of the volume sampling rate, pre-integrated volume rendering not only provides much higher quality for a given volume sampling rate but also allows larger sampling distances and accordingly higher performance while retaining the same visual quality. Therefore, pre-integration has become a standard technique for volume rendering.

Enhancing the basic ray casting shader with pre-integrated transfer functions requires only trivial modifications. Figure 4.8 shows the changes that have to be made to the original code of Figure 4.4. Ray setup, volume sampling, compositing, advancing the ray position, and ray termination remain unchanged. The only difference is in the transfer function lookup. Instead of directly indexing a 1D texture that stores the sampled transfer function, a lookup in the 2D pre-integrated transfer function table is made using the scalar values of the current and the previous sampling position as texture coordinates. Furthermore, the scalar value of the current iteration has to be saved to serve as the front scalar value in the next iteration. Note that the additions necessary for accurate ray termination presented in Section 4.2.3, including the opacity correction step, remain unaffected by these changes.

In this case, the single-pass ray casting approach is more efficient in terms of texture-sampling operations than slice-based volume rendering. Since the scalar value of the previous sample can be efficiently cached in a shader register or shader variable, only one lookup in the volume texture is necessary per sampling step. In contrast, a standard slice-based approach would require an additional lookup in order to retrieve the previous scalar value as well.

4.4.2 Isosurfaces

The display of isosurfaces is still the most prominent and widely used technique for visualizing volumetric data. In the optimal case, an isosurface will represent the important feature the user is interested in—the bone surface in a CT-scan, a shock front in a CFD data set, or simply the surface of a scanned object represented through a distance field.

The classical approach for visualizing isosurfaces is to extract a polygonal approximation to the surface by marching through all cells of the data set [132]. For large data sets and in case only an image and no explicit geometrical representation of the surface is required, however, ray casting is generally much more effi-

```
// Input textures
uniform sampler3D VOLUME;
uniform sampler2D TRANSFERFUNCTION;
...
void main(void)
{
    ...
    // Initialize scalar value
    float sf = texture3D(VOLUME, pos).r;

    // Advance sample position
    pos += dir;

    do {
        // Sample volume at current ray position
        float sb = texture3D(VOLUME, pos).r;

        // Look up color and opacity in pre-integration texture
        vec4 src =  texture2D(TRANSFERFUNCTION, vec2(sf, sb));

        // Perform front-to-back compositing
        gl_FragColor += (1.0 - gl_FragColor.a)*src;

        // Advance sample position
        pos += dir;

        // Save scalar for pre-integration
        sf = sb;

    // Test ray termination criterion
    } while (pos.x >= 0.0 && pos.x <= 1.0 && pos.y >= 0.0 && pos.y <= 1.0 &&
             pos.z >= 0.0 && pos.z <= 1.0);
}
```

Figure 4.8: Single-pass pre-integrated volume ray casting. Only changes with respect to the basic ray casting code given in Figure 4.4 are shown.

cient. Although isosurfaces could be also treated as a special case of direct volume rendering and rendered using, for example, specifically defined pre-integration tables [107], it is often much more convenient to describe them by the first-hit semantic that naturally describes the intersection of a surface with a viewing ray. Furthermore, this approach provides an explicit ray-surface intersection point and a surface normal that, amongst others, allows complex shading of the surface, such as self-shadowing or environment mapping.

During ray traversal an isosurface intersection can be detected by a sign change in the difference between the scalar value and the user-defined isovalue. Therefore, the color and opacity lookup and the compositing computation in the basic ray casting code is replaced by this test. As for pre-integrated rendering, either the scalar value or the difference to the isovalue can be stored between samples to avoid unnecessary texture lookups. Once an intersection has been detected, the

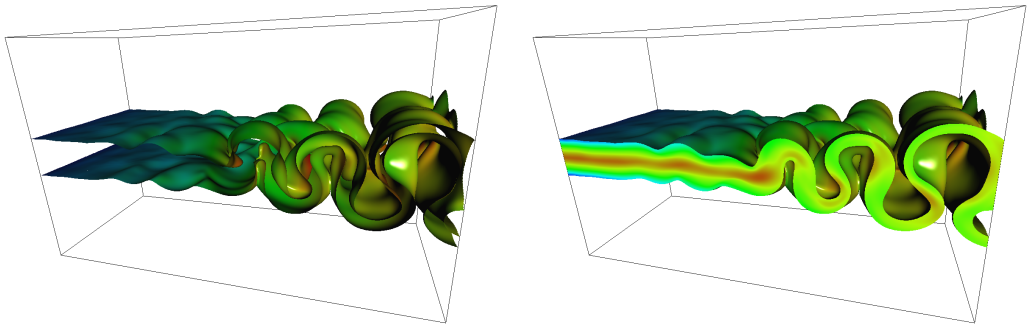


Figure 4.9: Isosurface renderings of the separation layer in a two-phase simulation of the breakup process of a liquid sheet for injection nozzle design [182]. Velocity magnitude of the flow field is color-coded on the surface. Left: Simple isosurface. Right: Closed isosurface.

ray traversal loop is terminated. Hit point refinement is employed to improve the accuracy of the intersection position. Assuming a linear variation of the scalar field between the two sampling positions linear interpolation can significantly improve the smoothness of the surface approximation [198]. Additionally, iterative bisection, as proposed by Hadwiger et al. [71], could be employed to improve the intersection point further. However, when using trilinear interpolation for reconstruction we did not notice any significant visual improvement over the simple linear interpolation approach.

For shading of the surface either precomputed gradients stored in an additional RGB 3D texture or on-the-fly computed gradients can be used. Since shading is computed only once for opaque isosurfaces and only visible surface points are shaded, the additional texture lookups and computations necessary for computing the gradient information are insignificant compared to the number of sampling steps typically necessary for finding the intersection. Given the explicit intersection position it is also straightforward to visualize additional scalar fields stored in 3D textures on an isosurface, as is shown in Figure 4.9.

Another effect that is difficult to achieve with a standard slice-based volume renderer but comes virtually for-free in GPU-based ray casting are closed isosurfaces. Here the term closed does not refer to holes in the actual isosurface due to insufficient sampling but rather to cases where the isosurfaces is not closed since it intersects with the boundary of the volume. An example from computational fluid dynamics is shown in Figure 4.9. Engineers often prefer such a closed representation since the 3D structure of the volume enclosed by the surface is much easier to comprehend.

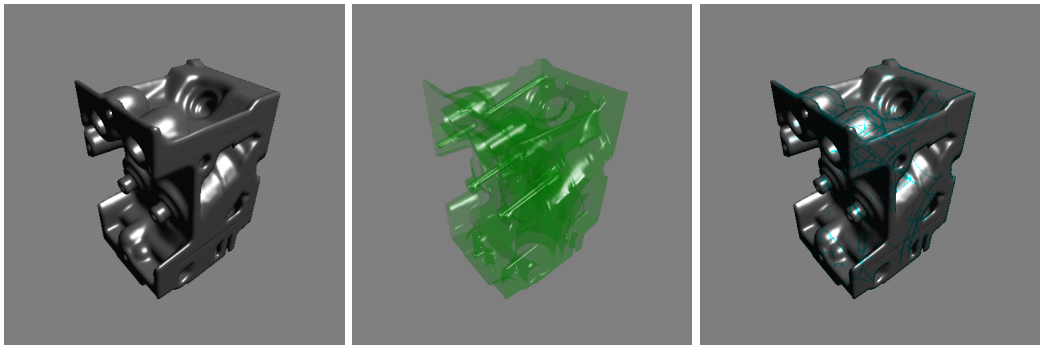


Figure 4.10: Different possibilities for displaying the hidden structure of an isosurface. From left to right: Isosurface, transparent isosurface, isosurface with hidden contours.[I]

A gap in the isosurface caused by an intersection with the bounding box is detected by checking the scalar value at the ray start position against the isovalue. A user-defined flag is used to define whether scalar values that are larger or smaller than the isovalue are considered to be interior of the isosurface. If thus a gap is detected, shading is computed for the intersection point and the shader execution is terminated. For shading, the surface normal is just the outward-pointing normal of the respective bounding box face.

Isosurface visualizations often suffer from self-occlusion; especially the closed isosurfaces defined above. One solution is to use transparent isosurfaces. Transparency, however, often results in cluttered images that are again hard to interpret. Another possibility is to use hidden contour lines defined where view vector and isosurface normal are approximately orthogonal. For both approaches ray traversal is not stopped at the first hit but the ray is traced through the whole volume or until a contour line is hit respectively. Both techniques are illustrated in Figure 4.10 and compared to a standard isosurface rendering.

Although an isosurface representation is often preferred over a semi-transparent volume rendering, in many cases it is desirable to have more context information available. In these cases it is also possible to combine semi-transparent direct volume rendering with (multiple) transparent or opaque isosurfaces. Here, the shaded isosurface is blended, using the same front-to-back compositing scheme, with the accumulated approximation to the volume rendering integral. Of course, care must be taken regarding opacity correction for the truncated segment in front of the isosurface. Some examples are show in Figure 4.11. Here, semi-transparent volume rendering provides context for the bone structures in focus.

[I] Data set courtesy of General Electric.

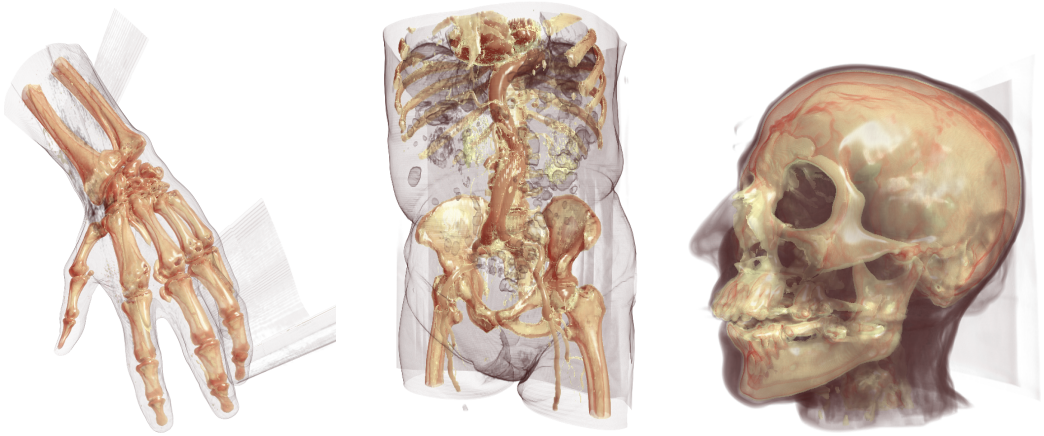


Figure 4.11: Various high-quality volume renderings of anatomical data sets[II] using a combination of opaque isosurface rendering and semi-transparent volume rendering to provide both focus (bone) and context (tissue).

4.4.3 Towards Photo-realistic Volume Rendering

Traditional optical models used in volume rendering, such as the emission-absorption model, do not account for important optical properties that are typically observed for real-world semi-transparent materials, such as glass or water. This includes effects like reflection, refraction, diffraction, scattering, selective chromatic absorption, chromatic dispersion, and polarization of light. However, depending on the application domain such effects are undesirable or at least unnecessary. Especially in scientific visualization, when dealing for example with data from medicine or engineering, refractive and diffractive effects are highly unwanted since they influence the perceived color values and may distort the desired effect of the applied transfer function; rendering the resulting images effectively useless for a qualitative analysis of the data. Furthermore, the effects of refraction or diffraction are hard to interpret correctly. On the other hand, there are also applications of volume rendering that aim for the photo-realistic rendering of volumetric objects, such as fluids, clouds, fire, and other gaseous or participating media.

An example how standard RGBα volume rendering fails in combining different colored semi-transparent layers in an optically correct manner is given in Figure 4.12. Since color accumulation only depends on a single opacity value and all intensity components of the light are attenuated equally, it does not account for selective chromatic absorption of light. In contrast, a spectral color model, such as the one that will be introduced in this section, will reproduce how the amount of

[II] Data sets are courtesy of the Department of Radiology, University of Iowa (Hand), M. Meißner, Viatronix Inc. (Abdomen), and the University of North Carolina at Chapel Hill (Head).

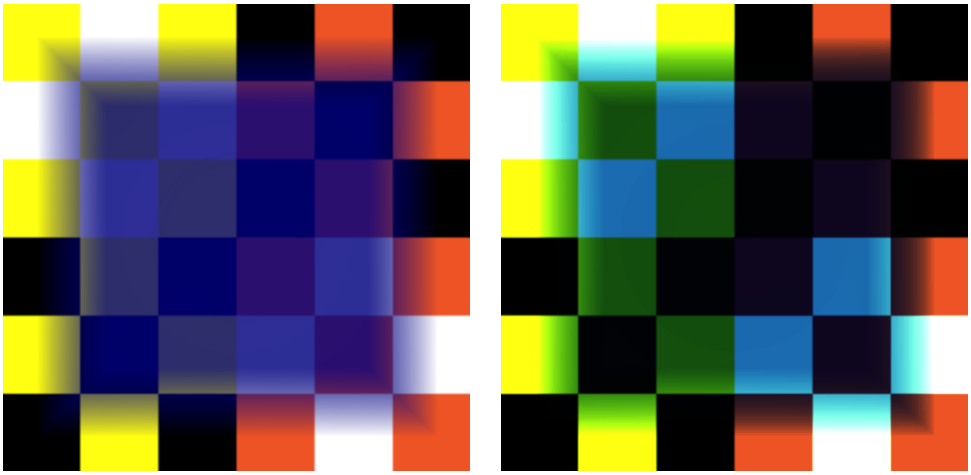


Figure 4.12: Comparison of volume ray casting based on traditional RGBα blending (left) and the spectral Kubelka-Munk model (right). Note how the appearance of the background in the latter case is affected by selective chromatic absorption when it is seen through the blue colored cube.

light absorbed by a real medium varies over the spectrum. The semi-transparent homogeneous blue cube shown in the right image of Figure 4.12 absorbs most wavelengths of the red spectrum, thus the yellow squares of the background appear to be green when seen through the cube and the red parts of the background completely loose their color information. Neither effects are reproduced using α-compositing, as is shown in the left image. The RGB transfer function that has been used to generate this image was computed from the spectral properties of the cube's material and the opacity transfer function has been adapted to match the appearance of the cube shown on the right.

In computer graphics the importance of spectral color models for photo-realistic rendering has been recognized for a long time [73, 59, 39]. Consequently, a number of alternatives and extensions to the traditional RGBα model have been developed and deployed for rendering high-quality images [158, 92, 203, 215, 67]. In volume rendering, however, only few publication can be found that are concerned with wavelength-dependent effects or take refraction into account.

Noordmans et al. [151] were the first to demonstrate that spectral volume rendering can produce physically more realistic visualizations when compared to direct volume rendering based on a traditional RGBα transfer function model. Later, Bergner et al. [10, 11] presented a spectral volume rendering system, which facilitates interactive data exploration by introducing a post-illumination approach, and considered different volume rendering techniques, such as ray casting, volume

splatting and Fourier volume rendering [12].

Abdul-Rahman and Chen [1] employed the Kubelka-Munk theory [189, 114, 112] of radiation transport in turbid media to simulate selective chromatic absorption and simplified scattering in volume rendering and show that it offers results that are more realistic than traditional RGBα. Their approach further allows for post-illumination and is based on real measured spectral material properties.

The visualization of refraction in volume rendering has been studied, for example, by Rodgman and Chen [173, 174]. They describe a ray casting approach that includes continuous refraction in volumetric data sets. In contrast to surface-based refraction that describes refraction of light on the boundary of two homogeneous materials of constant refractive index, as has been studied by Li and Mueller [127], they assume a continuous function describing the refractive index of an inhomogeneous material. Examples for applications that require a continuously changing refractive index field are the visualization of inhomogeneous mixtures of fluids, impurities in transparent materials, or fluids of varying density, such as hot air.

The approach presented here combines the work of Abdul-Rahman and Chen on spectral volume rendering [1] and Rodgman and Chen on refractive volume ray casting [173, 174] in a GPU-based approach. This volume shader for spectral volume rendering and refractive effects was developed and first published in collaboration with Magnus Strengert and Ralf Botchen [201].

A Short Introduction to Kubelka-Munk Theory

Kubelka-Munk theory provides a different approach to the light transport in participating media. The Kubelka-Munk model is basically a two-flux approach to the general radiation transport theory, which considers two radiation fluxes that pass through a continuous medium in two opposite directions—one forward along the ray direction, which represents an average of all forward-scattered light, and one backward against the ray direction, which represents the average of all backward scattering. While an early version of the model assumed that the medium is completely homogeneous [114], Kubelka later extended the model to inhomogeneous layers of paint-like turbid substances, where each elementary layer is assumed to be isotropic and completely diffuse [113].

Consider a volume consisting of n homogeneous elementary layers of thickness d. For each layer the discrete Kubelka-Munk model assumes an average *reflectance* R and *transmittance* T, which are characterized by the Kubelka-Munk scattering and absorption coefficients S and K, respectively. According to Kubelka [112] reflectance and transmittance are related to scattering and absorption coef-

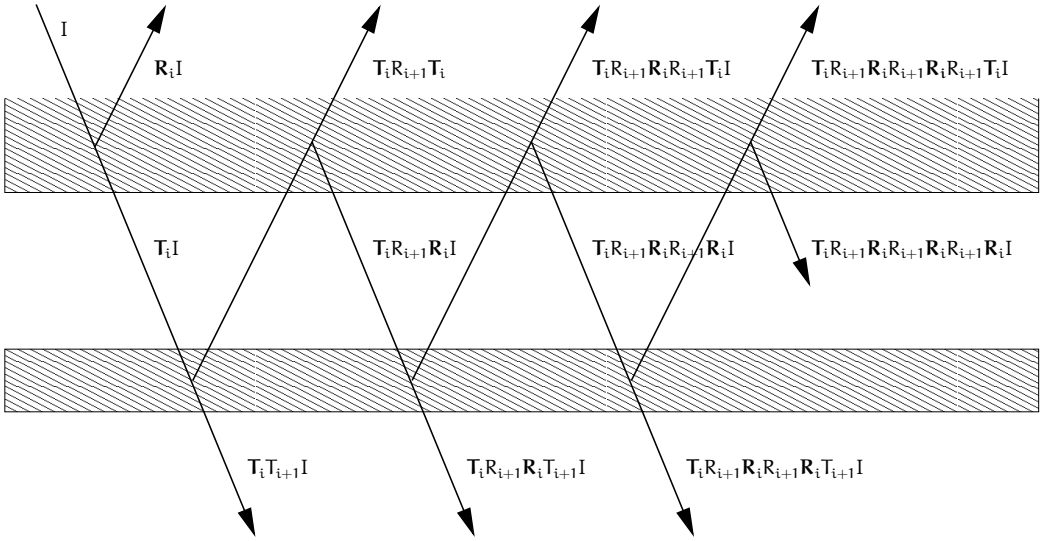


Figure 4.13: Accumulation of diffuse light contributions according to the Kubelka-Munk model.

ficients as:

$$R = \frac{\sinh(bSd)}{a\sinh(bSd) + b\cosh(bSd)}\,, \tag{4.17}$$
$$T = \frac{b}{a\sinh(bSd) + b\cosh(bSd)}\,,$$

where

$$a = \frac{S+K}{S}, \quad b = \sqrt{a^2 - 1}. \tag{4.18}$$

Note that the absorption and scattering coefficients K and S are actually functions of the wavelength and can be derived from real material properties [68, 9]. Hence, similar to the transfer functions used in traditional volume rendering, for each scalar value of the volume, a corresponding set of spectral absorption and scattering coefficients S_λ and K_λ must be defined that maps data values to material properties.

For a given incident light ray of intensity I the forward light flux, or the transmitted light, from layer $i-1$ passes through layer i with the portion $T_i I$, then through the next layer with the portion $T_i T_{i+1} I$, and so on. Meanwhile, a portion of flux of the amount $T_i R_{i+1} T_i I$ will be reflected from layer $i+1$ and passes backwards through layer i again. The infinite process of interaction between layers, which is illustrated in Figure 4.13, can be considered as a kind of one-dimensional

radiosity computation [42]. The sums of the two infinite series of reflectance and transmittance define the composited reflectance $\mathbf{R}$ and transmittance $\mathbf{T}$ for each slab of the volume.

Let $\mathbf{R}_i$ and $\mathbf{T}_i$ be the accumulated reflectance and transmittance of layer 0 to layer i and let R_{i+1} and T_{i+1} be the sampled reflectance and transmittance of layer $i+1$. Similar to the discrete approximation of the volume rendering integral in Section 4.1.2, a recurrence relation can be derived for the composite reflectance and transmittance of the layers 0 through $i+1$:

$$\mathbf{R}_{i+1} = \mathbf{R}_i + \mathbf{T}_i R_{i+1} \sum_{j=0}^{\infty} (\mathbf{R}_i R_{i+1})^j = \mathbf{R}_i + \frac{\mathbf{T}_i^2 R_{i+1}}{1 - \mathbf{R}_i R_{i+1}} \tag{4.19}$$

and

$$\mathbf{T}_{i+1} = \mathbf{T}_i T_{i+1} \sum_{j=0}^{\infty} (\mathbf{R}_i R_{i+1})^j = \frac{\mathbf{T}_i T_{i+1}}{1 - \mathbf{R}_i R_{i+1}}. \tag{4.20}$$

These front-to-back Kubelka-Munk compositing equations assume the light source to be co-located with the view point. On the same basis a similar set of equations can be derived for light entering the volume opposite of the viewer. In contrast to color compositing, however, Kubelka-Munk compositing results in a reflectance and transmittance distribution that has to be lit with an appropriate light spectrum to produce the spectral color information.

Beside being a standard model in the paint and paper industry [238], the Kubelka-Munk model has also found a number of applications in computer graphics. It has been used for rendering of metallic patinas [42], the simulation of realistic pigment mixing [68], watercolor effects [32], and the realistic modeling of wax crayon [179] and oil paints [9].

Spectral Volume Rendering based on the Kubelka-Munk Theory

Deploying the Kubelka-Munk model in a volume raycaster is not difficult. The basic structure of ray traversal, volume sampling, classification, and accumulation remains essentially the same. However, the wavelength dependent behavior of the media has to be taken into account for spectral effects. For each wavelength a reflectance value according to the reflectance and transmittance encountered along the sampled path in the volume has to be computed. In other words, the classification by the transfer function lookup and the volume integration based on alpha blending is replaced by a lookup of the Kubelka-Munk coefficients S_λ and K_λ for the sampled material or scalar value in a material table texture, the evaluation of the Equations (4.17) for the reflectance and transmittance, and the Kubelka-Munk compositing as defined by Equations (4.19) and (4.20) for all wavelengths

of the spectrum. This is accomplished by sampling the spectrum of visible light at equidistant positions in the range from 400nm to 700nm at 10nm intervals, resulting in a total of 31 distinct wavelengths that have to be considered independently. The final result of this accumulation process is a spectral reflectance map $\mathbf{R}_n$, which represents the interaction of the volume with incident light.

To complete the process, the contributions of the background have to be added. Assuming a completely opaque background layer, a reflectance map $R_B(\lambda)$ can be obtained from an RGB image using an algorithm by Glassner [58] that computes the spectral representation of a RGB value under controlled lighting conditions. Note that since there is an unlimited number of possibilities to define the spectrum of an RGB color triplet and only a limited number of frequencies are represented in the sampled spectrum, RGB values are not accurately reproduced in this approach. However, the approximation is sufficiently close for our application.

The final spectral reflectance map $\mathbf{R}_I(\lambda)$ of the image can then be computed according to Equation (4.19) by compositing $R_B(\lambda)$ and the accumulated reflectance of the volume $\mathbf{R}_n(\lambda)$ as

$$\mathbf{R}_I(\lambda) = \mathbf{R}_n(\lambda) + \frac{\mathbf{T}_n^2(\lambda) R_B(\lambda)}{1 - \mathbf{R}_n(\lambda) R_B(\lambda)} \; . \tag{4.21}$$

Finally, as a last step, the computed reflectance map has to be lit with an appropriate light source to produce the final color values of the image. This is achieved by applying, for example, the spectrum $L_{D65}(\lambda)$ of the CIE standard illuminant D65 representation of natural daylight [234]. Which results in a spectral representation of the image:

$$I_\lambda = \mathbf{R}_I(\lambda) L_{D65}(\lambda) \; . \tag{4.22}$$

Finally, the spectral image has to be converted to RGB color space for displaying the result. This is accomplished by converting the color spectrum I_λ to the XYZ color space using the tristimulus color matching functions, $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$, defined for the CIE 1964 standard colorimetric observer [234] by

$$I_{XYZ} = \sum_\lambda I_\lambda(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)) \tag{4.23}$$

and subsequent transformation to RGB space

$$I_{RGB} = M_{XYZ \to RGB} I_{XYZ} \tag{4.24}$$

using the color conversion matrix $M_{XYZ \to RGB}$ given by Glassner [58].

The images rendered by the algorithm outlined above in general look flat and unrealistic, especially for very opaque materials. Like in RGBα volume rendering

this can be improved by adding a shading term. Since the Kubelka-Munk model assumes isotropic light scattering that obeys Lambert's cosine law the local reflectance coefficient can be multiplied by a diffuse shading term without violating the underlying model.

Although it is theoretically possible to treat all 31 wavelengths in parallel in a single fragment shader program, there are several drawbacks that militate against this approach. Most importantly, it would even for fairly large sampling steps exceed the maximum limit of executed fragment shader instructions per fragment imposed by many GPU architectures. Therefore, it is beneficial to treat each wavelength separately using a multi-pass solution. Since the summation of color matching functions and the following color transformation are linear operations, additive blending can be used for accumulating the contributions of the 31 rendering passes in a floating point render target.

Furthermore, although a straightforward implementation of Equations (4.17) could be done in the fragment shader, this is not very efficient since it would require a lot of costly shader instructions like square root computations and the evaluation of hyperbolic sine and cosine. Fortunately, when the thickness d of the layer is known in advance, which is the case for ray casting with a constant step length, R and T can be precomputed for a finite set of scalar values and wavelengths and stored in a two-dimensional texture map. Since both functions are bounded between 0 and 1 it is possible to store these values with high precision in a floating point texture.

Another important property of spectral rendering methods is the possibility of re-illuminating the rendering result by changing only the power spectrum of the light source [12, 1]. This allows to generate different visual impressions without repeating the ray casting process. Changing the illumination of the volume thus provides another degree of freedom for interactive exploration of the data.

Raycasting of Refractive Volumes

Besides selective chromatic absorption there are other important effects that can be observed for real transparent or translucent media. The perhaps most obvious are refraction caused by variation in the optical density of the material and total internal reflection that occurs in certain cases on the transition from an optically denser to an optically lighter material. Another effect closely related to refraction is chromatic dispersion. This is a phenomenon that arises from the variation of the refraction coefficient of the material over the spectrum of light. Since different wavelengths are refracted to different degrees, an achromatic light ray is split into rays of chromatic light that follow different paths through the medium. This phenomenon is typically associated with prisms.

Although the Kubelka-Munk model assumes that the entire volumetric do-

main of colorant has the same refractive index [113], this should not prevent the introduction of refraction in the model in an approximative manner. First, adding refraction in the Kubelka-Munk model is not in any way less accurate than adding refraction in the traditional volume rendering integral [174], since the traditional volume rendering integral is based on the Lambert-Bouguer law [234], which also assumes that the entire volumetric domain is of the same refractive index. Second, if there were refraction between different elementary layers, changing the direction of a flux according to the relative refractive index between the two layers is more accurate than no directional change of the flux. The reason of this is obvious as the dominant direction of transmittance in the phase function of a volume is largely determined by the field of the refractive index.

Implementing refractive volume rendering is straightforward with the presented ray casting approach, requiring only few modifications. The first is an additional refraction coefficient that has to be stored in the material texture for each of the 31 wavelengths of the spectral rendering model. The second is the update of the current ray traversal direction. Similarly to storing the scalar value of the previous sample in the case of pre-integrated volume rendering the refraction coefficient obtained for the last sample can be kept in a shader variable. Then, for each integration step a new ray direction $\mathbf{d}_i'$ is computed according to Snell's law as

$$\mathbf{d}_i' = \eta \mathbf{d}_i - \left(\eta(\mathbf{d}_i \cdot \mathbf{n}_i) + \sqrt{1 - \eta^2(1 - (\mathbf{d}_i \cdot \mathbf{n}_i)^2)}\right)\mathbf{n}_i \,, \tag{4.25}$$

where $\eta = \eta_{i-1}(\lambda)/\eta_i(\lambda)$ is the relative refractive index computed from the refractive indices associated with the current and the last sampling point following the concept of a continuous refractive index field as defined by Rodgman and Chen [174]. Furthermore, $\mathbf{n}_i$ is the volume gradient at the current sampling position, if necessary re-oriented to point in the direction opposite to $\mathbf{d}_i$. Both $\mathbf{d}_i$ and $\mathbf{n}_i$ are assumed to be normalized.

If the angle between the volume gradient and the incoming direction of the ray exceeds the critical angle $\psi_t = \arcsin \eta^{-1}$ the incoming ray is reflected from the material interface, rather than being refracted. This total internal reflection is indicated by a negative value below the square root in Equation (4.25). In this case $\mathbf{d}_i$ is set to the reflected ray direction

$$\mathbf{d}_i' = \mathbf{d}_{i-1} - 2(\mathbf{d}_i \cdot \mathbf{n}_i)\mathbf{n}_i \,. \tag{4.26}$$

Unfortunately there are some problems with total reflections. It is possible and in the case of real data sets not unlikely that very long light paths can occur when a ray is reflected multiple times inside the volume. Such problems are typically caused by noise and inaccurate gradients or numerical inaccuracies in the computation. This not only makes the computation very slow but also may exceed

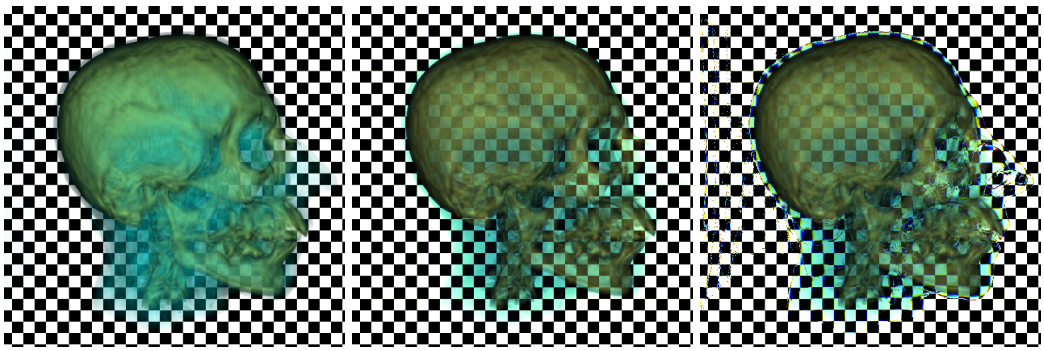


Figure 4.14: Comparison of RGBα volume ray casting using a pre-integrated transfer function (left), spectral volume ray casting based on the Kubelka-Munk model (middle), and spectral volume ray casting including also refractive effects (right).

the maximum number of executed fragment shader instructions. As a solution we have limited the maximal number of total reflections that are allowed when traversing a ray through the volume to a total of five. This effectively contains the number of iterations in the sampling loop and in fact does not introduce significant error, since multiple reflections are not easily understood by the observer and in the case of volume rendering typically lead to noisy images.

In general, refractive volume rendering is much more susceptible to low-quality normals due to noise in the scalar data as it is the case, for example, in shading isosurfaces. The fact that erroneous gradient information does not only affect the local lighting computation but also the direction of the ray and accordingly all subsequent sampling points has a much higher impact on rendering quality and spatial as well as temporal aliasing than it will have on shading. Rodgman and Chen [174] therefore propose extensive anisotropic nonlinear diffusion filtering to smooth the input scalar volume. As a trade-off between continuous smooth gradients and distortion of the volume data, we use gradient data that is computed in a preprocessing step using a 3D Sobel operator. Additionally pre-smoothing of the volume data and post-smoothing of the gradient field using Gaussian filter kernels can be applied depending on the noisiness of the input volume.

Figure 4.14 shows three renderings of a CT data set using the same basic rendering parameters, but three different ray casting techniques. The leftmost image shows the result of standard pre-integrated volume rendering using a transfer function derived from the spectral material properties chosen for generating the other two images. Note that only the color components of the RGBα transfer function were automatically calculated, while the opacity values were chosen to match the appearance of the spectral renderings as close as possible. The other two images were rendered based on the Kubelka-Munk model with 31 samples of

the spectrum of visible light as described in the previous section. In contrast to the rightmost image the middle image does not include refraction. But distortion of the background image as well as the inner structures of the volume and refraction fringes due to dispersion are clearly visible in the last image. Both effects significantly help to depict the three-dimensional nature of the data set. Parts of the volume that were hardly visible in the first two images due to their high transparency can now be clearly identified. On the other hand, artifacts due to noise in the data set are much more apparent when rendering with refraction as it is easy to notice in the left part of the image.

4.5 Acceleration Techniques

In Section 4.2.1 a straightforward and easy implementation of GPU-based ray casting was presented. However, although this implementation is simple and elegant, it is rather inefficient. There are a number of optimizations that can provide significant advantages in terms of improved interactivity. Therefore, in the following a number of optimizations will be discussed that either reduce the overall instruction count of the ray casting shader or are concerned with certain idiosyncrasies of GPU architecture.

Besides such programmatic optimizations, the presented single-pass ray casting approach can be also easily extended to incorporate various algorithmic acceleration techniques, such as early ray termination, adaptive sampling, and empty space skipping. These will be also discussed in this section.

4.5.1 Programmatic Optimizations

As for every programmable processing unit programmatic optimizations play a crucial role for achieving optimal performance. In the following some possible optimizations of the inner ray casting loop will be discussed. The aim of this section is not to present optimizations for a specific hardware but general procedures that apply to all modern GPU architectures.

Efficient Ray Termination

Although it is convenient to write down the ray termination as straightforward as it was done in the code shown in Figure 4.4, it is worthwhile to take a closer look at the costs associated with evaluating the termination criterion. There are several factors that affect this cost and therefore have to be taken into account for an efficient implementation. It turns out that the ray termination criterion can be evaluated much more efficiently when the particular structure of the problem

and the idiosyncrasies of the GPU are taken into account. Since most GPUs, at least those by ATI/AMD and NVIDIA, feature native hardware support for automatic saturation, i.e. clamping to the $[0, 1]$ interval, of ALU instruction results [3, 154], checking the current sampling position against the unit cube and subsequent termination of the traversal loop can be accomplished very efficiently by only a few assembly instructions. Exploiting color saturation modifiers and vector-valued condition code tests the solution proposed here boils down to only three (`NV_fragment_program2`) assembly instructions—a saturating register move, a subtraction, and the data-dependent loop exit instruction.

```
...
MOV_SAT temp.xyz, pos;
SUBC temp.xyz, temp, pos;
BRK (NE.xyzz);
...
```

However, this is hard to achieve in a high-level shader language, such as GLSL, since the structure of the generated low-level code and the actual number of instructions is strongly compiler-dependent and varies for different vendors, driver versions, and hardware profiles. Unfortunately, at the moment non of the high-level shader languages does support inlining of assembly instructions. The closest match that can be realized using GLSL is given below:

```
...
} while (clamp(pos, vec3(0.0), vec3(1.0)) - pos == vec3(0.0));
...
```

This provides a 10 to 20% performance gain over the straightforward implementation of the ray termination criterion.[III]

On the other hand, it could be argued that it would be even more efficient to replace the dynamic ray termination criterion by precomputation of the number of necessary ray integration steps before entering the integration loop [153] or to compute the ray parameter value of the ray exit point and compare the current value against that value [110]. However, the first would not allow any kind of adaptive sampling. While this would be possible with the latter approach, both approaches do not allow for changing ray directions, which, for example, is necessary when rendering refractive volumes as in Section 4.4.3.

Furthermore, in terms of shader instructions, the optimized termination criterion described above is nearly as efficient as checking against the current ray parameter value. The latter requires two shader assembly instructions; including the conditional branching instruction.

[III] Measured on NVIDIA GeForce 6800 GT and NVIDIA GeForce 8800 GTS for pre-integrated direct volume ray casting of a 256^3 volume data set and an integration step size of half a voxel distance.

Unrolling parts of the loop has been also proposed as a solution to reduce the cost of ray termination [49]. The unavoidable overshooting that is caused by this method is countered by using a texture border of zero color and opacity; thus, no contributions are accumulated for ray positions outside the volume. This method, however, suffers from the same artifacts caused by premature termination as discussed in Section 4.2.3. Furthermore, it does not work with pre-integrated transfer functions as the assumption of a continuously changing scalar field, which can be locally approximated by a linear function, is no longer valid.

4.5.2 Algorithmic Accelerations

The flexibility of the ray casting approach provides the possibility for the realization of a number of obvious and not so obvious acceleration techniques. The former includes standard techniques like early ray termination or adaptive sampling in the ray direction, which are not easily transfered to slice-based rendering.

Early Ray Termination

Early ray termination [123] is a standard optimization technique for volume ray casting. The basic idea is to stop the ray traversal as soon as the contribution of further samples will be negligible for the overall ray casting result. In general, this will be the case as soon as the accumulated opacity saturates, i.e. reaches a value close to one. Typically a user-specified opacity value $\alpha_{\text{ert}} \lesssim 1$ is used as a threshold. The additional test for early ray termination can then be simply added to the ray termination criterion.

```
...
} while (clamp(pos, vec3(0.0), vec3(1.0)) - pos == vec3(0.0) &&
         gl_FragColor.a < ertThreshold);
...
```

The effect of early ray termination, however, is obviously data, transfer function, and view dependent. While early ray termination may greatly reduce the number of ray traversal steps in case of isosurface-like renderings, for highly transparent transfer functions the additional overhead imposed by the early ray termination check may even prove detrimental to rendering performance.

Figure 4.15 gives an example how early ray termination affects the number of ray traversal steps. Where the ray hits the isosurface-like bone structures ray integration can be stopped rather early, which reduces the overall number of ray traversal steps and texture sampling operations necessary for rendering the image by 21% on average. This results in a average increase in rendering performance by 50%. This is not a particular good example for early ray termination. Since large

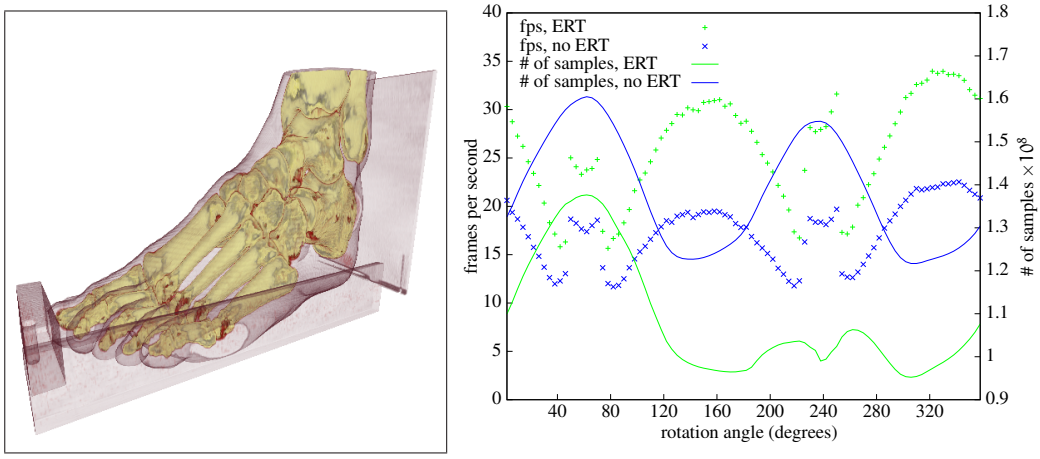


Figure 4.15: Efficiency of early ray termination. Left: Pre-integrated direct volume rendering of a 160×430×183 8-bit CT-scan data set.[IV] Right: The number of 3D texture sampling operations with and without activated early ray termination ($\alpha_{\text{ert}} = 0.98$) and the corresponding performance in frames per second.[V]

parts of the volume are mapped to highly transparent values by the chosen transfer function, the speedup gained by terminating the ray integration early is only marginal. However, this test shows that the overhead for adding the additional test to the termination criterion is rather small.

4.5.3 Adaptive Sampling

While pre-integration ensures sufficient sampling of the transfer function, it does so by assuming that the scalar field varies only linearly between two consecutive sampling positions. Therefore, the sampling step size has to be chosen sufficiently low in order to ensure adequate sampling of the scalar field and accordingly the volume rendering integral. However, using a global integration step size that allows proper sampling of high-frequency regions of the data set does waste computations due to oversampling uniform or slowly varying regions of the volume. Fortunately, and in contrast to slice-based volume rendering, the sampling step size can be chosen freely for each ray and for each individual sampling step in case of ray casting. Therefore, the step size can be dynamically adapted to the local characteristics of the scalar field, allowing faster traversal of homogeneous regions in a data set, which in turn can provide a drastic reduction in sampling operations.

[IV] Data set courtesy of the Department of Radiology, University of Iowa.
[V] NVIDIA GeForce 8800 GT (G92), 512MB.

Since it is not feasible to determine the minimum sampling distance on-the-fly during shader execution adaptive sampling relies on an additional data structure that defines for a given sampling point the minimum isotropic distance to the next volume position that ensures appropriate sampling of the volume. This so-called *importance volume* or *step size oracle* is computed in a preprocessing step and stored in an additional 3D texture. Typically, the resolution of this texture will be chosen much lower than the actual volume in order to keep the memory overhead low. Then, in addition to the scalar value lookup another lookup is performed to retrieve the appropriate sampling distance for the following iteration step.

Computing the importance volume is commonly based on local variations of the scalar field and a user-defined error tolerance [49]. Röttger et al. [177] propose to use a step size criterion that is based on the second derivative of the scalar field but do not provide any further detail on the actual computation nor a justification for this particular choice. Bergner et al. [13] use—according to their theoretically derived sampling criterion as described in Section 4.1.4—an importance volume computed from the maximum value of the gradient magnitude in a local voxel neighborhood.

Adaptive sampling can be further combined with the idea behind early ray termination [33]. The simple observation is: The higher the accumulated opacity, the lower will be the contribution of further samples and accordingly the error that is introduced by increasing the sampling step size. Thus, choosing the sampling step size proportional to the accumulated opacity can increase the rendering speed without too much impact on image quality.

Just as for early ray termination, the advantages gained by adaptive sampling depend strongly on the input data as well as the view and rendering parameters. There is also no general rule for balancing the overhead of the additional operations against the possible reduction of traversal iterations. Whether the additional effort is beneficial or not has to be decided on a case by case basis.

4.5.4 Exploiting Frame-to-Frame Coherence

When visualizing a volumetric data set many regions of the volume are often not of interest to the user, e.g. the soft tissue when visualizing the bone structure in a medical CT data set, and therefore are set to completely transparent in the transfer function. The same applies when rendering isosurfaces or isosurface-like representations. In this case large parts of the volume will also not contribute to the final rendering result. Although adaptive sampling allows to traverse those regions in larger steps it does not take into account the fact that even more computations and sampling operations can be saved by skipping them entirely. However, whether a region of the volume can be assumed to be completely transparent or empty and, thus, does not contribute to the final rendering result depends on the scalar data as

well as the transfer function.

Many different approaches have been proposed to skip empty space during ray traversal but not all of them are easily adapted to the GPU. Most of those techniques rely on some kind of regular or hierarchical space partitioning data structure, like octrees [123, 71], bounding volumes [4], BSP trees [128], or distance transforms [241, 30, 196] to distinguish between empty and nonempty regions and to avoid traversing and sampling empty voxels that do not contribute to the volume rendering integral.

Although hierarchical space-partitioning techniques provide by far the best speed-up, especially in the case of isosurface ray casting, they also have disadvantages. They require additional storage for the meta data, cannot be efficiently mapped on current graphics processing units, or the data structure may even have to be rebuilt for every change of the transfer function.

Another group of empty-space–skipping methods exploits the fact that during user interaction or when rendering animation sequences spatio-temporal coherence for rays shot through the volume is very high. Thus, using information from previously rendered images can be used to skip empty regions by directly *leaping* to the first data voxel or ray sampling position that exhibits significant data values. Ray traversal is then started at this point, reducing the volume sampling cost significantly.

Empty-Space Leaping by Reprojection

The first to describe such empty-space–leaping techniques, which exploit the spatio-temporal coherence in successively rendered animation frames to accelerate volume ray casting, have been Gudmundson and Randén [63] and Yagel and Shi [236] in the early 1990s. While the solution shown by Gudmundson and Randén is limited to parallel projections, the introduction of an intermediate *coordinates buffer* that stores the object-space coordinates of the first nontransparent voxel encountered during ray traversal by Yagel and Shi allows also for perspective projections. By reprojecting the content of the coordinates buffer of the previously rendered frame using point-splatting according to the change in viewing parameters they obtain an estimate of the ray starting positions for computing the subsequent image. This ray casting step can then take advantage of the coordinates buffer content to skip the empty space not contributing to the final rendering result and start directly at the first relevant sampling positions.

Several extensions and improvements to the original algorithm have been proposed in the meantime. Wan et al. [213] use a cell-splatting approach, reprojecting voxels instead of points and combine it with precomputed distance-based empty space skipping. Another approach was demonstrated by Yoon et al. [240]. Instead of transforming the point coordinates they project the rays into the coordinates

buffer of the previous animation frame in order to rapidly find intersections with isosurfaces. A recent approach by Lakare and Kaufman [117] exploits ray coherence instead of inter-frame coherence to build the space leaping structure. The basic empty-space–skipping approach—based on ray coherence, frame-to-frame coherence, and space partitioning—has been also used in traditional ray tracing of polygonal objects. Furthermore, the concept of reprojection is also useful for generating stereo image pairs.

The most important advantage of these reprojection techniques is the fact that they do not rely on an additional data structure that requires time-consuming preprocessing and consumes extra memory. Furthermore, as will be described in the following, an empty-space–leaping approach can be fitted easily in the rendering pipeline of GPU ray casting and, as will be shown, can directly benefit from the native data models and the parallel processing power of the GPU [103].

Mapping Empty-Space Leaping on Graphics Hardware

The three phases of the empty-space–leaping algorithm outlined above—initialization of the coordinates buffer, ray casting and update of the coordinates buffer, as well as the reprojection of the coordinates—can be easily integrated into the GPU-based ray casting system. Three off-screen render targets are used to store intermediate information, i.e. one buffer that stores the final ray casting result and two that are used as coordinates buffers in a ping-pong fashion. All render targets have floating point precision.

The accelerated ray casting proceeds then as follows. Before the first frame is rendered the content of the coordinates buffer is initialized with the object-space positions corresponding to the front faces of the volume's bounding box. Then for each frame, the start positions for the ray casting are read from the coordinates buffer and two render targets are written: the usual color image of the volume ray casting and the updated coordinates buffer storing the positions of the first relevant ray sampling points, in the following referred to as the *hit points*, encountered during ray traversal. For consecutive frames, the previously stored hit points are reprojected, i.e. transformed regarding to the new viewing parameters.

The graphics processor is perfectly suited for this kind of point projection technique as it is designed for fast vertex processing. Besides, the vertex processing units are mostly idle in a GPU-based ray casting approach as it is heavily rasterization-bound and the only geometry involved so far is the proxy-geometry represented by the faces of the volume bounding box. Thus, the overhead imposed by projecting the hit points onto the image plane can be regarded insignificant compared to the overall computational cost of ray casting. Furthermore, the necessary depth sorting of the reprojected points comes virtually for free.

Initialization Before ray casting the first image in an animation sequence, the start points for the rays cast through the volume have to be initialized in order to start the process with a well-defined state. As no further information is available, yet, the intersections between the rays and the bounding box of the volume have to be used, similar to the standard algorithm. Thus, the front faces of the bounding box are rendered into the texture bound to the coordinates buffer, mapping interpolated object-space coordinates to RGB color values.

Reinitialization of the ray start positions is also necessary if, for example, the window is resized, thus invalidating the render targets, or large changes of the volume or view parameters, e.g. isovalue, transfer function, or large camera movements, have been performed by the user. However, moderate changes in the volume parameters are compensated by the conservative estimate taken for the ray start positions. Thus, reinitialization is only necessary if, e.g., the isovalue is changed in very large steps or a completely new transfer function is loaded.

Ray Casting The single pass ray casting shader has to be slightly modified to accomplish the space leaping. For each ray, the start position for volume traversal is no longer given by the interpolated object-space corners of the volume's bounding box but has to be read by a texture lookup from the coordinates buffer holding the reprojected hit points of the previously rendered frame.

During ray casting, the position of the first sampling point contributing to the final image is written to the coordinates buffer. For opaque isosurface rendering this corresponds to the intersection point of the ray with the surface, whereas when using semi-transparent volume representations the first sampling point is stored, which corresponds to a scalar value that is not mapped to a completely transparent color. To cope with inaccuracies during the following reprojection step that would lead to inexact results or even artifacts when, for example, the foremost isosurface would be missed, the hit point has to be moved slightly back along the ray before it is written to the buffer. A conservative estimate of the number of samples to step back to ensure that no significant data voxel is missed could be $n_{max} = \left\lceil \sqrt{\Delta_x^2 + \Delta_y^2 + \Delta_z^2}/d \right\rceil$. However, this is a very conservative estimate. Stepping back one to three sample lengths has turned out to be sufficient in most cases.

For rays completely missing the volume, i.e. no nontransparent voxel has been encountered during ray traversal, the ray-bounding box exit point is used as the hit point. Note that coordinates buffer elements lying outside the volume's projection also have to be initialized to some meaningful value to ensure that those points do not interfere with the following transformation and projection step. This is crucial in order to avoid problems caused by such points being projected inside the volume's image-space footprint during reprojection. Therefore, the coordinates

buffer is initialized before ray casting with a coordinate value that is guaranteed to lie outside the view frustum after applying the view transformation by rendering a viewport filling polygon. Then, view frustum culling ensures that those points do not have an effect on the projected point set.

Reprojection The third step is to obtain a new estimate for the initial ray sampling positions for the next image by projecting the coordinates buffer content into the image plane specified by the viewing parameters of the new frame. As the point coordinates are stored in object space it is sufficient to initialize the OpenGL modelview and projection matrix according to the new view parameters and to feed the positions through the geometry pipeline to compute the new coordinates of their screen space projection, i.e. the origin of the ray that is expected to hit them. Only the necessary points are rendered, i.e. those corresponding to pixels inside the screen aligned bounding rectangle of the projected volume, to save on vertex processing. A vertex shader is used to set the color of the resulting fragment to the input object space coordinates of the respective hit point position.

A separate vertex array containing dummy vertex positions is used as input for the projection. The dummy vertex positions, in fact, have to represent the texture coordinates with which the input texture bound to the previously written coordinates buffer has to be sampled in the vertex shader to get the actual vertex coordinates. Then, the vertex position retrieved by the texture lookup is projected to screen-space via the modelview-projection transform. Additionally, each fragment of the projected point is assigned the object space coordinate of the point as the RGB color value before it is written to the other one of the ping-pong buffers, serving as the ray starting position for the following ray casting. Note that reprojection could be realized much more efficiently on recent graphics hardware using geometry instancing instead of the dummy vertex coordinates array.

Reprojection Errors

After reprojection most pixels have assigned the correct starting position for the subsequent ray casting step. However, there are still two problems that have to be addressed. First, pixels containing no information about reprojected positions can occur in the image. This is because the screen-space positions of the reprojected points do not necessarily coincide with the integer framebuffer pixel locations but instead are assigned to the pixels nearest to their normalized window-space coordinates. Thus, no space-leaping information is available and we have to start a full-scale ray casting from the first meaningful position, i.e. the intersection of the ray with the volume boundary. The second problem is also related to the discrete nature of the framebuffer. Although in theory multiple reprojected points that are mapped to the same pixel location should be sorted in the correct depth order, it is

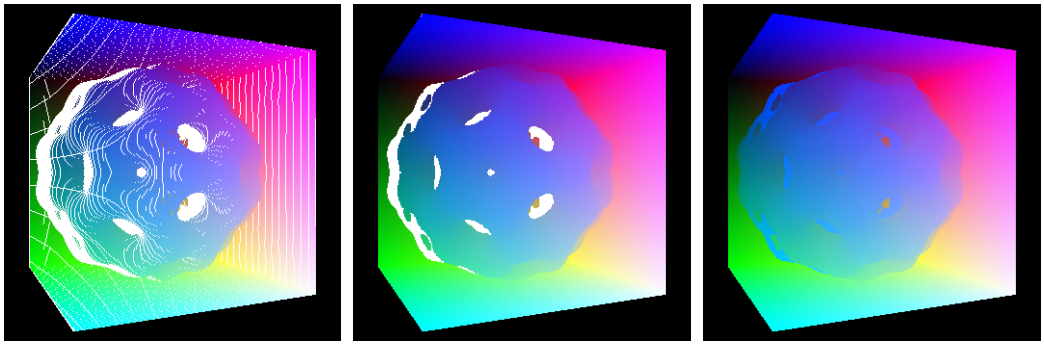


Figure 4.16: Illustration of the effects of the two error correction steps. Left: Coordinates buffer after reprojection. Middle: Splatting extended points. Right: Content of the coordinates buffer after both steps.

still possible to end up with a wrong estimate for the initial ray position in certain cases. Due to the described discretization problems and the nature of the perspective transformation it is likely that values of points that should have been actually covered by points lying closer to the viewer are still visible. Hence, the wrong start positions would be chosen and parts of the volume would be missed during ray casting. Several solutions for this problem have been proposed, including the use of point-splatting and cell-splatting methods [213, 236].

These splatting approaches exploit the fact that projecting points with a screen-space footprint larger than a single pixel can solve the occlusion problem as long as the points are spread densely over the image plane. In this case depth sorting guarantees that for overlapping points always the most conservative estimate, i.e. the point nearest to the viewer, is chosen. This corresponds to the depth sorting proposed by Lakare and Kaufman [117]. Choosing a point size of two or four we have not noticed any artifacts due to wrong depth ordering in our test cases. Also, rendering points that are larger than one pixel poses no significant overhead.

Although rendering enlarged points also significantly reduces the number of holes in the image there are often some pixels left for which no information is available. These pixels have to be identified and preset to a meaningful value. A very conservative measure is to set them to the intersection of the ray with the volume's bounding box. This is accomplished by initializing the destination buffer prior to reprojecting the points by rendering the bounding box front faces.

The images shown in Figure 4.16 illustrate the effect of the two error correction steps for an isosurface rendering. The left-most image shows the content of the coordinates buffer after reprojection using point primitives of size one. White areas represent pixels for which no space-leaping data is available. The few outliers in red on the front of the isosurface depict pixels with erroneous space-

Table 4.1: Performance of the GPU-based ray casting with and without empty-space leaping (ESL). Shown are the average framerates measured while rotating the volumes according to the views shown in Figure 4.17 multiple times about the vertical axis. The sampling step size h is given with respect to the shortest edge of the voxels.[VI]

data set	size	h	no ESL	ESL	speed-up
Foot (4.17a)	160×430×183	0.4	6.7	18.8	2.7
Head (4.17b)	256^2×225	0.5	7.7	17.1	2.2
Abdomen (4.17c)	512^2×174	0.7	3.9	10.4	2.7
Foot (4.17d)	160×430×183	0.8	9.7	13.2	1.4
Aneurysm (4.17e)	512^3	0.5	3.6	6.2	1.7
Aneurysm (4.17e)	512^3	1.0	7.0	11.9	1.7
Abdomen (4.17f)	512^2×174	0.7	5.6	6.8	1.2

leaping information due to incorrect depth sorting. The middle image shows the result of reprojection using points of size four. All pixels that suffered from incorrect occlusion now have valid information. The right image shows the result after updating the remaining holes with positions on the volume's front faces.

On the other hand, the assumption about frame-to-frame coherence is obviously only correct for small differences in the viewing parameters for consecutive images. Artifacts can occur when violent interaction and low framerates are involved. Fortunately, splatting of extended points provides a certain degree of self-correcting behavior, i.e. artifacts that arise from incorrect space-leaping information are mostly corrected automatically, as soon as the view differences in consecutively rendered images fall below a certain level. This is a result of the propagation of depth information in the vicinity of projected points caused by their enlarged screen space footprint.

Effectiveness of Empty-Space Leaping

The effectiveness of the presented empty-space–leaping technique is obviously strongly dependent on the particular data set, the used transfer function, and other view parameters. For evaluating the accelerations that can be achieved for typical scenarios a comparison of GPU-based ray casting with and without empty-space leaping has been conducted for several medical volume data sets. Table 4.1 sum-

[VI] NVIDIA GeForce 7800 GTX, 512^2 viewport.

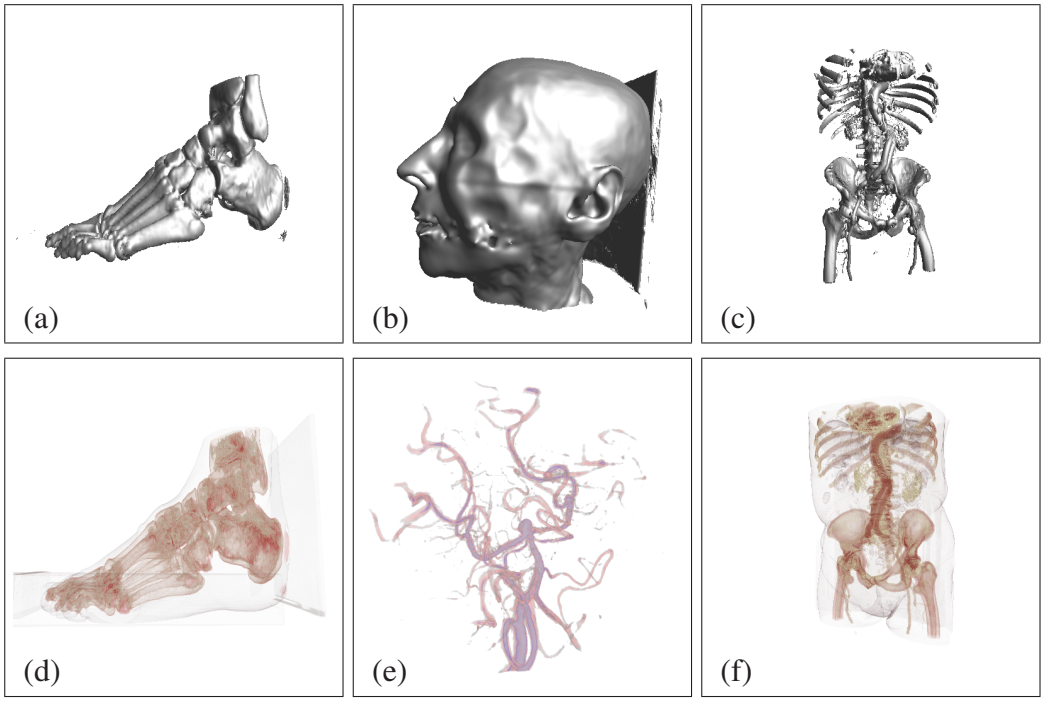


Figure 4.17: The data sets used for evaluating the effectiveness of empty-space leaping for both isosurface rendering (top row) and pre-integrated volume rendering (bottom row).[VII]

marizes the results. For all measurements the sampling step size was chosen to be at least as small as the minimal slice distance of the volume data set. All images were rendered to a 512^2 viewport according to the views shown in Figure 4.17. For the actual measurements the average framerate when rotating the volumes about the vertical axis have been taken.

As can be seen, a speed-up of more than a factor of two can be achieve for isosurface renderings. When rendering the isosurface of the Abdomen data set shown in Figure 4.17c without empty-space leaping an average of 177 sampling steps per pixel was necessary until the surface was hit. Using empty-space leaping this number is reduced to an average of only 25 steps, requiring only ten steps for more than 60% of the pixels. In the direct ray casting approach without space leaping only 1.8% of the rays could be terminated after only ten steps.

Unfortunately, the empty-space–leaping acceleration is not very effective for semi-transparent volume renderings if most parts of the volume contribute to the image, as shown in the last four rows of Table 4.1. A speed-up of only 20% could be achieved for the Abdomen data set if the skin is shown as in Figure 4.17f.

[VII] Abdomen and Aneurysm data sets courtesy of M. Meißner, Viatronix Inc.

4.6 Selective Supersampling

Due to the discrete sampling of the image plane, ray casting generates images with noticeable staircase artifacts predominantly at silhouettes, feature lines and regions of high gradient magnitudes. These aliasing artifacts are most apparent in images of static views and negatively affect the otherwise high quality provided by the described volume ray casting approach. A common approach to cope with this problem is to supersample the image plane. Modern graphics hardware provides built-in support for this kind of *full-scene antialiasing* (FSAA) for application-independent usage. The techniques used are based on sampling the area covered by a single pixel in image space multiple times and combining the results to the final color. The number of additional samples and their spatial arrangement are manifold and directly influence the image quality. Finally, all supersampling algorithms boil down to a performance versus quality trade-off, since shooting additional rays result in higher quality but also in higher computational costs.

Unfortunately, this built-in functionality cannot be utilized if rendering to textures bound to an off-screen buffer is required. Although the creation of and the rendering to multi-sampled off-screen render buffers is possible using the `EXT_framebuffer_multisample` OpenGL extension, rendering to multi-sampled textures or mixing multisampled and non-multisampled draw buffers in a framebuffer object is unfortunately not support at the moment.

However, rendering to textures is required for most multi-pass algorithms or, for example, for the space-leaping acceleration described in Section 4.5.4.

Therefore, to further improve the visual quality of the generated images, the ray casting algorithm is required to provide an explicit antialiasing mechanism. In the following an approach will be presented that allows both full-scene super-sampled antialiasing as well as selective supersampling only for regions requiring antialiasing. The presented technique has been developed in collaboration with Magnus Strengert and Simon Stegmaier [103].

While full-scene antialiasing based on supersampling requires every pixel in image space to be refined, selective supersampling first analyzes the vicinity of each pixel and generates additional rays only for fragments that possibly suffer from aliasing artifacts. A simple classification criterion is based on the differences in color values of a fragment to its four-neighborhood compared to a user-defined threshold. If the magnitude of any of the four differences exceeds the given threshold, additional rays are generated and the fragment's color value is replaced by the average accumulated color of those rays, while in the other case the original color value is retained unchanged. This allows for improving the image quality in the most significant regions without the need to refine the complete image space at a higher sampling rate.

Selective supersampling is realized as an additional render pass. In the first

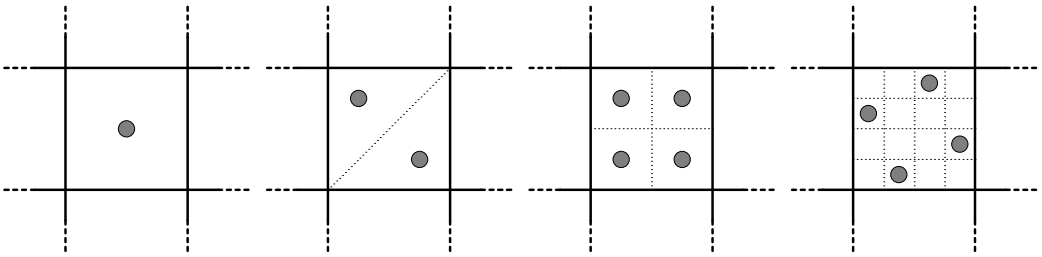


Figure 4.18: Sampling patterns used for antialiasing. From left to right: Single sample, 2× supersampling, 4× supersampling, and 4× rotated-grid sampling. Solid lines indicate pixel boundaries.

pass the normal ray casting is performed to an off-screen render buffer. This buffer provides the basis for the classification in the second pass where the raycasting process is repeated. In this pass, for each fragment its four neighbors are examined and additional rays are generated if necessary.

For the distribution of the additional sampling rays three different sampling patterns comparable to patterns used for FSAA on recent graphics hardware have been investigated. Besides two regular sub-pixel schemes for two-fold and four-fold supersampling, as a third sampling pattern rotated-grid sampling [155] has been employed. Rotated-grid sampling uses a diamond-shaped arrangement for optimized sub-pixel coverage. This proves to be superior to a regular pattern since each row and column of the sub-pixel is exactly covered by a single sample point, as illustrated in the comparison of all implemented sampling patterns in Figure 4.18.

In order to determine ray directions for the additional rays, a linear interpolation scheme based on the ray directions of the current fragment and the directions of two of its neighbors as depicted in Figure 4.19 has been employed. The weights controlling the linear interpolation are specified via a set of texture coordinate values, allowing for arbitrary sampling positions. So more advanced patterns other

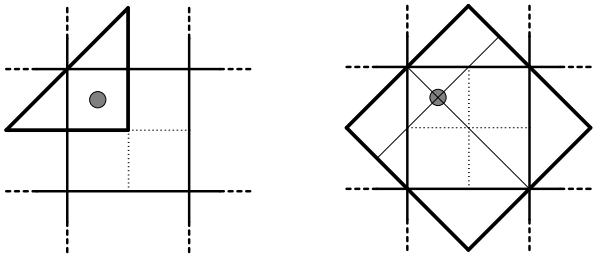


Figure 4.19: Alternative schemes used for determining the sub-pixel ray directions. Left: linear interpolation, right: bilinear interpolation.

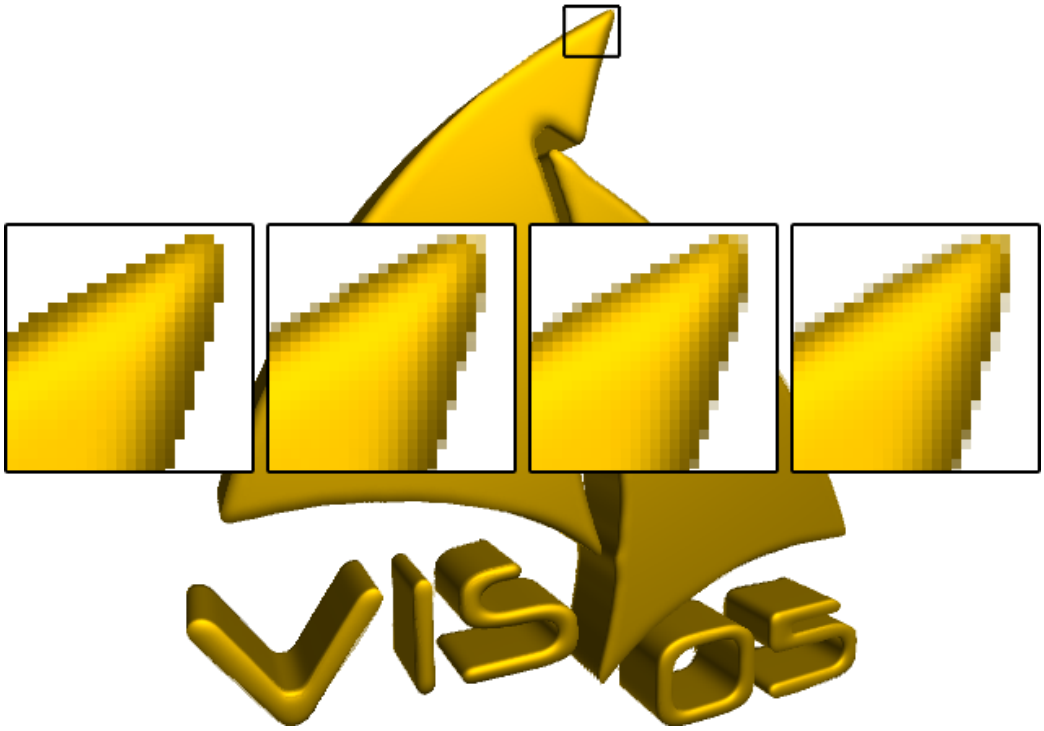


Figure 4.20: Comparison of antialiasing quality. From left to right: no antialiasing, 2× super-sampling, 4× regular super-sampling, and 4× rotated-grid sampling.

than the already described can be used as well. Since the four direct neighbors need to be looked up anyway, bilinear interpolation could also be used, which provides better coverage of the four-neighborhood.

The influence of the different sub-pixel sampling patterns on the final image and the original unaltered reference are depicted in Figure 4.20 for the isosurface rendering of a distance field. As expected four-fold super-sampling outperforms the two-fold approach, but as already mentioned above, the increase in quality is traded for speed. Rendering the shown isosurface representation without antialiasing results in 30.7 fps on an NVIDIA GeForce 7800 GTX GPU. When using two-fold full-scene supersampling this decreases to 12.6 fps and finally casting four rays results in a framerate of 9.4 fps. The high performance loss in the first case is primarily due to the overhead imposed by additional texture lookups required for determining values of the four direct neighbor pixels. Accordingly, there is only a relatively small drop in performance when using higher subsampling rates since no further texture lookups are necessary. Reducing the number of refined pixels with selective supersampling does not automatically increase performance for the case of volume ray casting. As all modern GPUs process fragments in larger

groups, processing many noncoherent fragments poses a considerable overhead that diminishes the advantages of selective refinement.

4.7 Evaluation

Comparing volume rendering approaches or just different implementations of the same technique is often difficult since volume rendering results depend strongly on the test data set, the current viewing parameters, and other rendering parameters, such as sampling step size, isovalue, or transfer functions. Even if those parameters are fixed a performance comparison has to take into account the size of the volume, the used viewport size, the fraction of the viewport covered by the image of the volume, and whether a static view or an animation is rendered. An exhaustive list of parameters, which have to be considered for reproducible volume rendering was compiled by Williams and Uselton [229]. Furthermore, performance figures depend on the underlying hardware and the level of software optimization. Therefore, comparing a given solution to already published results is often impossible because in many cases one or more of the above mentioned parameters is unclear.

In the following an evaluation of the rendering performance and rendering quality of GPU-based ray casting will be presented. It will be compared to standard slice-based GPU-assisted volume rendering in terms of image quality and rendering performance within the general volume rendering framework introduced in Section 4.3.

4.7.1 Measuring Rendering Quality

In case of visualization techniques, the term quality has two connotations: *intelligibility* and *fidelity*. Intelligibility describes the suitability of a particular visualization for a specific task, e.g. its suitability for medical diagnosis or operational planning; in other words whether the employed method provides an optimum of information relevant for the analysis by the domain specialist. However, this type of visualization quality is often not assessable without profound knowledge of the particular application area and is often more a question of choosing the adequate rendering modality or transfer function instead of the difference between different realizations of similar visualization techniques.

Here, we will focus on the technical quality or fidelity of the images, which can be evaluated irrespective of a particular application. The technical quality of an image is typically assessed by comparing it to a given *ground truth* image that is considered the optimal, artifact-free rendering result. In most volume rendering papers, images are compared visually by juxtaposition or by using difference

images. Although visual comparison cannot provide a quantitative evaluation of image difference, due to the lack of objectiveness and consistency by a human observer, it is nevertheless useful to provide an overview of the spatial distribution and the qualitative magnitude of the difference between two images. This cannot be represented by a single numerical value provided by a mathematical metric. Therefore, in this section both, an aggregate full-reference image difference metric for quantifying the magnitude of deviation of a rendering result from the ground truth and side-by-side comparisons of rendering results, are used for comparing GPU-based ray casting as proposed above and standard hardware-assisted slice-based volume rendering.

Many metrics for comparing color images of volume rendering results have been proposed [229, 94, 232]. The simplest and most often used just compare luminance information or Euclidian distance in RGB space in the form of *mean square error*, *signal to noise ratio*, or other statistical metrics. However, human color perception is not linear. Therefore it is better to use color difference metrics that work in a perceptually (approximately) uniform color space, such as CIE-$L^*a^*b^*$ or CIE-$L^*C^*H^*$. For these exist standardized color difference metrics, such as the CIE ΔE metrics. However, in real scenes these simple metrics provide an unsatisfactory measure of the actually observed image difference. Therefore perceptual image difference metrics have been proposed that take the sensitivity of the human visual system to structural information, color contrast, and intensity into account.

In the following a perceptual image difference metric proposed by Yee [239] is used for comparing volume-rendered images against a given reference image. This metric is based on a multi-scale computational model of the human visual system that predicts the maximum luminance and chrominance deviations that can be tolerated to consider two images equivalent, taking the human visual system's loss of sensitivity at high spatial frequencies, high contrast levels, and high background illumination into account. Both a luminance error and a chrominance error is provided by this metric. In the original implementation of Yee only a binary predicate whether a pixel is perceptually different from its reference based on an error threshold is computed. This has been modified so that instead the numerical value by which this threshold is exceeded is returned. A single image fidelity figure is then obtained by taking the average over all pixels of the image.

4.7.2 Rendering Quality

As stated above the possibility of quantitative assessment of image quality stands or falls with the availability of a reference image that can be considered the true volume rendering result. However, in general such an image can be only obtained by analytic integration of the volume rendering integral, which is not possible for

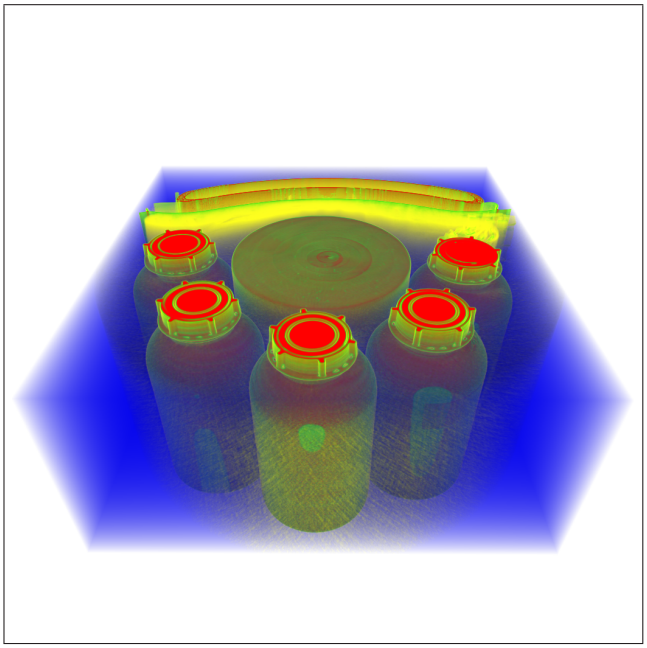


Figure 4.21: Test scene for evaluation of image fidelity and rendering performance.

sampled volume data and arbitrary transfer functions, or approximated by expensive adaptive numerical integration. As a compromise, in the following reference images are used that have been generated using ray casting and slice-based rendering with extreme oversampling. A sampling step size or slice distance corresponding to taking 32 samples per voxel with respect to the largest voxel extent and 19 samples per voxel with respect to the smallest voxel extent has been chosen. The slice-based renderer uses 32-bit floating point blending for high-quality results.

As a test data set a $512^2 \times 442$ 12-bit CT scan of a colon phantom[VIII] is used. To fully exploit the available accuracy of the 16-bit 3D texture the input data range of $[0, 3967]$ has been remapped to the full $[0, 2^{16} - 1]$ range. The transfer function has been chosen such that both opaque, isosurface-like structures as well as highly transparent regions are present in the volume. Hence, both sampling artifacts and inaccuracies due to limited blending precision are likely to occur if the volume is sampled too low.

This way two sets of reference images have been created. A series of 180 images has been rendered at a resolution of 1024^2 pixels for different viewpoints by rotating the camera in steps of four degrees at a fixed distance around the

[VIII] Data set courtesy of Michael Meißner, Viatronix Inc., USA.

vertical scene axis. Figure 4.21 shows the data set from the viewpoint of the first test image. The volume's center is located in the coordinate origin and it has been rotated by 52 degrees about the positive x-axis. A view typical to volume visualization has been chosen. A perspective projection with a 60 degree field-of-view has been used and the camera was located 1.4 units off-center looking down the negative z-axis.

Image fidelity and rendering performance has been compared for GPU-based volume ray casting and slice-based volume rendering with viewport-aligned slices. For slice-based rendering framebuffer formats with 8-bit color depth per RGB channel as well as float precision render targets with 16-bit and 32-bit float precision per color channel have been employed to study the effect of higher precision blending on image quality. For reasons of fair comparison the basic GPU-based ray casting implementation shown in Figure 4.8, i.e. no optimizations or quality-increasing enhancements, such as early ray termination, adaptive sampling, or special treatment of the final ray segment, as discussed in Section 4.2.3 have been used, and a basic 3D slice-based volume renderer have been compared. In both cases pre-integrated classification has been used. So there should not be any principle advantage for either of both implementations. Furthermore, comparing against both sets of reference images guarantees that there is no unfair bias towards any of the methods. The same viewing conditions as for the reference images have been used. The sampling rate was set to one sample per voxel.

Figure 4.22 shows the results of the experiments conducted on a standard PC equipped with a 2.4GHz Intel Q6600 Core2 CPU, 4GB RAM, and a NVIDIA 8800 GT (G92) based graphics card with 512MB of graphics memory. The topmost and the middle graph show the average perceptual chrominance and luminance difference to the reference images according to the image difference metric by Yee for each of the rendered views. Only pixels contained in the projection of the volume's bounding box have been considered when computing the average. Note that as for every statistical average measure, just like for example peak-signal-to-noise ratio, the absolute magnitudes of the errors are not as important as their relative magnitudes. The third graph shows the average rendering performance in frames per second that has been achieved for each of the views for the four tested rendering modes.

There are several interesting observations that can be made in these graphs. First, the fidelity of the images generated by ray casting is typically much higher than those rendered with the slice-based implementation regardless whether compared to the reference images created by ray casting or texture slicing. Of course, each method performs best when compared to the reference image set rendered by the same approach. The differences, however, are directly proportional to the differences of the reference image sets which are additionally shown in both error plots. The spatial distribution of the chrominance errors is shown in Figure 4.23.

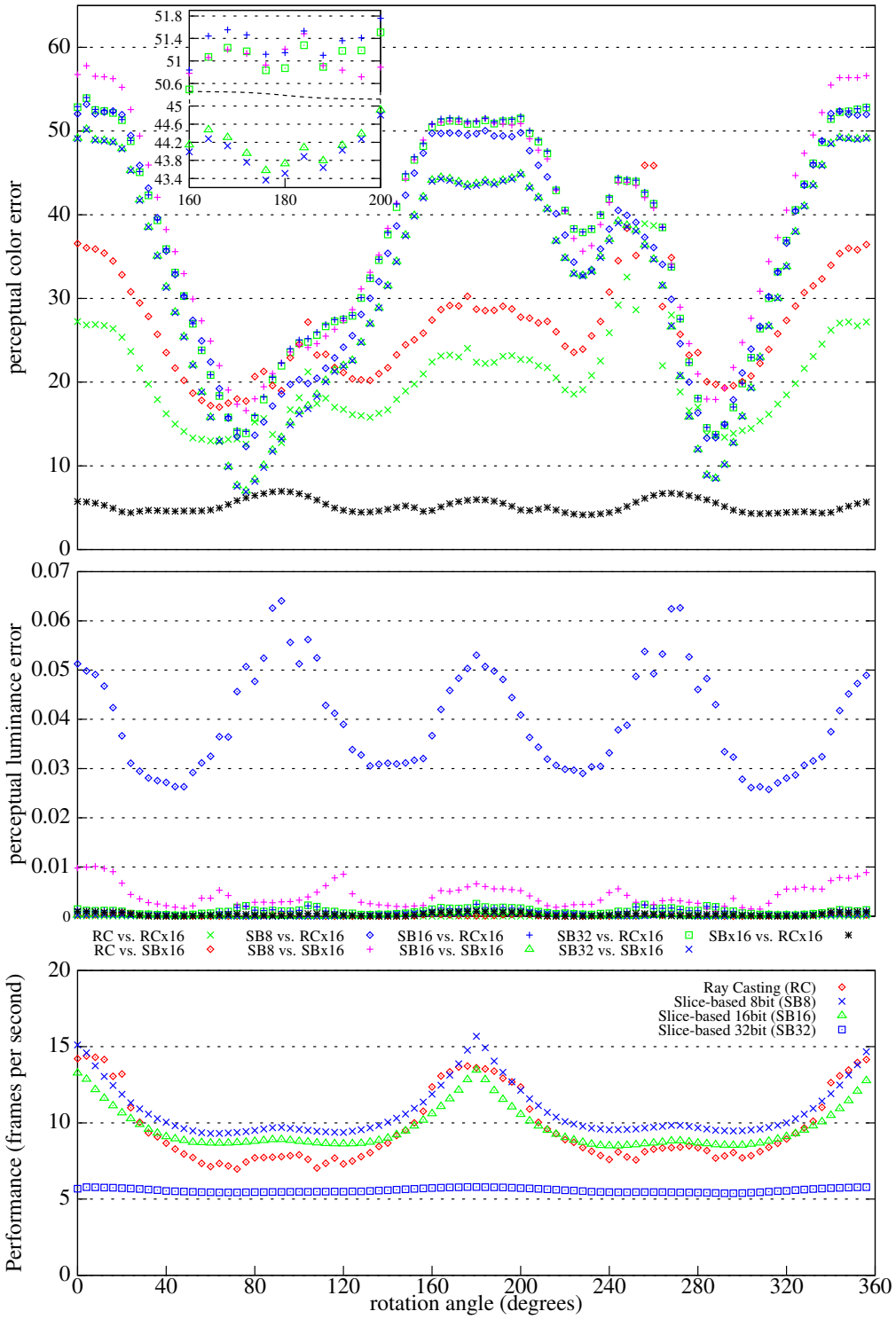


Figure 4.22: Comparison of image fidelity and performance for GPU-based volume ray casting (RC) and slice-based volume rendering (SB).

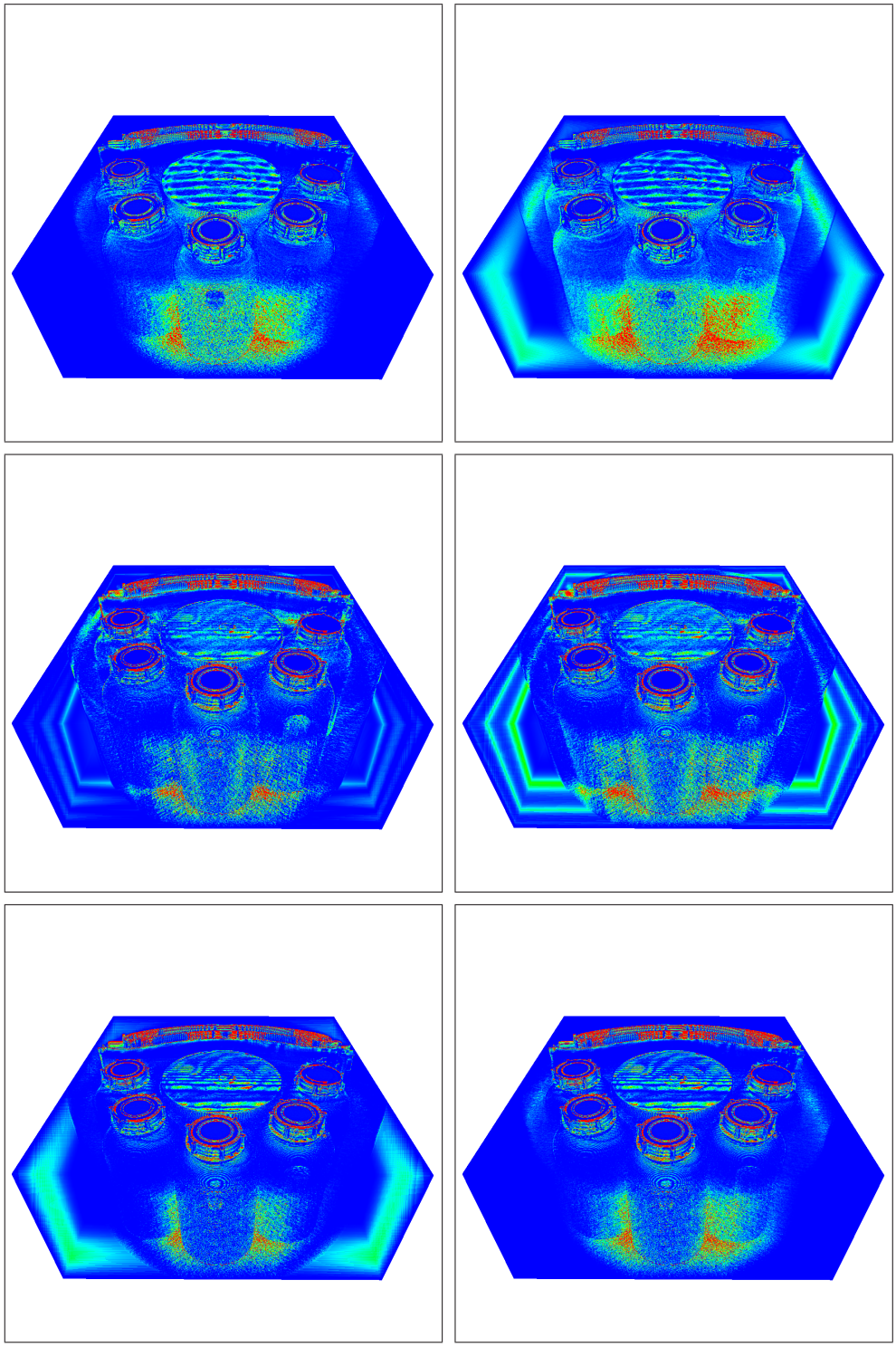


Figure 4.23: Spatial distribution of perceptual chrominance error for the first frames of the image sequences rendered to produce the graphs in Figure 4.22.

The left column shows the comparison to the ray casting reference while the right columnt shows the comparison to the slice-based reference. From top to bottom ray casting, slice-based rendering with 8-bit blending, and slice-based rendering with 32-bit floating point blending is shown. The color scale ranges from blue—no error—to red—large error.

As expected sampling artifacts can be noticed on the surfaces of the isosurface-like structures. Furthermore, slicing artifacts due to the nonequidistant sampling in slice-based rendering can be observed at the left and right boundary of the volume. Second, the difference between 16-bit and 32-bit float precision blending is only marginal in this example. This is further highlighted by the inset in the chrominance error plot showing a magnified representation of high error regions. In contrast, 8-bit blending has a significant impact on image quality. Although this might not be apparent in the chrominance error plot where it seemingly outperforms higher precision blending at least for the ray casting reference. While the luminance error is negligible for ray casting and slice-based rendering with 16-bit and 32-bit floating point blending regardless of the reference it is compared against, it is approximately an order of magnitude larger for 8-bit blending, which leads to differences that are quite noticeable in the rendered images. This indicates that the limited precision of 8-bit blending is problematic in highly transparent areas, where very small opacity contributions are not sufficiently reproduced in the quantized values, leading to significant errors in the luminance channel. Third, as can be seen from the performance graphs, in terms of rendering speed there is not much difference between GPU-based ray casting and slice-based rendering using 8-bit and 16-bit float render targets. However, 32-bit blending introduces a considerable overhead which results in a 50% decrease in the average rendering performance. This is an indication that ray casting and slice-based rendering using 8-bit and 16-bit float render targets are limited by texture sampling and interpolation. Depending on the view direction the number of sampling operations varies and has a direct relationship with the achieved rendering performance. On the other hand, slice-based rendering with 32-bit precision floating point blending is clearly limited by the much more expensive blending operation. In this case the number of samples has only a minor effect on rendering times.

4.7.3 Rendering Performance

In this section the scalability of GPU-based raycasting with regards to volume data size and viewport resolution will be analyzed and compared to slice-based rendering. Again, all presented results were generated on the previously described PC, no special acceleration techniques have been employed, and pre-integrated classification has been used. For this test, the measurements were run on two different midrange graphics cards, the NVIDIA GeForce 8800 GT (G92) based card used

in the previous section and an AMD/ATI Radeon HD 3870 based graphics board. Both cards feature 512MB of graphics memory.

As test data a 8-bit encoded functional approximation of the electron density inside a C_{60} Buckminsterfullerene[IX] has been used, which has been resampled to seven different resolutions ranging from 256^3 to 768^3 voxels. All volumes should thus fit into the available texture memory. Note that the 768^3 volume could not be rendered on the ATI card and is thus only measured for the NVIDIA GPU.

The same measurement procedure as in the previous section has been employed. Figure 4.24 shows the resulting average framerates for rotating the volumes the full 360 degrees about the vertical scene axis. That way, texture caching effects and other view dependent factors should be effectively eliminated. The view has further been chosen in a way that as much of the viewport is covered by the volume as possible but only small portions of the volume are clipped by the viewport borders when rotating the volume. Each of the different sized volumes has been rendered into viewports of varying resolutions of up to 1024^2 pixels. The sampling distance has been set to approximately one sampel per voxel. All other rendering parameters have been kept constant.

This data confirms the observations made in the previous test. Ray casting is slightly slower than slice-based rendering with 16-bit floating point precision blending. Especially for large viewports it is evident that both approaches are texture limited, i.e. the overall performance is limited by texture access and trilinear interpolation bandwidth. In view of which it must be noted that for the selected view, ray casting on average performs 6% more sampling operations than slicing due to the nonequidistant sampling of the slice-based approach. However, this does not fully account for the differences. Albeit, in theory, the overhead for setup and rasterization of the proxy geometry is much higher for slice-based volume rendering, the costs for dynamic flow control in the ray casting shader program seems to outweigh this benefit by far. So for reasonably sized viewports ray casting is approximately 25% slower than slice-based rendering with 16-bit blending on the ATI GPU and comparable in performance on the NVIDIA GPU. Thus, all in all, both approaches perform equally well in this case.

On the other hand, as has been shown by Bitter et al. [15], 16-bit floating point blending is not sufficient for high-quality volume rendering. They show that the theoretical minimum precision requirements for computations in a volume rendering pipeline for 12-bit volume data is a fixed point format with 16 integer and 16 fractional bits. Thus, even the 16-bit float representation, which provides only 10 mantissa bits, might be not sufficient for accurate compositing of very small opacity contributions. Therefore, another series of measurements has been

[IX] Data set courtesy of O. Kreylos of the Center for Image Processing and Integrated Computing at UC Davis.

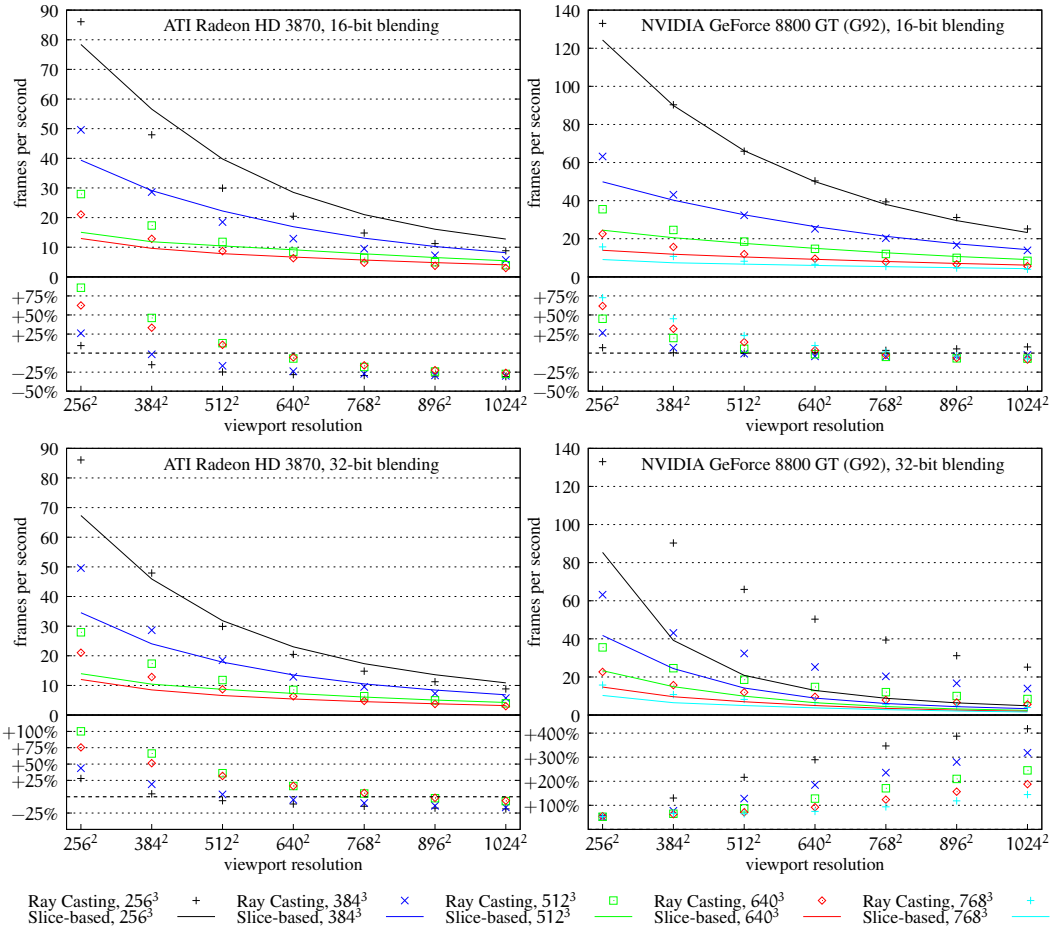


Figure 4.24: Comparison of rendering performance for GPU-based volume ray casting and slice-based volume rendering depending on viewport resolution, volume size, and blending precision. In each case the lower graphs shows the relative performance of GPU-based ray casting compared to the slice-based approach.

performed with 32-bit-blended slices. The results are shown in the bottom row of Figure 4.24. In this case the behavior of the two GPUs in question is more distinct. While the Radeon HD 3870 loses some additional 25% for small volumes the overall behavior remains the same. Ray casting gains only on a rather small scale. In contrast, on the GeForce 8800 GT rendering is clearly limited by the expensive blending operations. Ray casting is consistently 50% to 400% faster and, what is even more important, this advantage grows as the viewport resolution is increased.

4.8 Conclusion

As has been shown in the previous sections, on contemporary commodity hardware basic single-pass ray casting in terms of rendering speed performs equally well as slice-based volume rendering using viewport-aligned slices. The limiting factor on rendering performance for both approaches is texture sampling, i.e. memory bandwidth and trilinear interpolation performance. The theoretical advantage of reduced overhead for rasterization of proxy geometry is outweighed by the cost of dynamic flow control in the ray casting shader program. Similarly, although in case of pre-integrated classification only one lookup in the volume texture per integration step is necessary, since compared to traditional texture-slicing techniques the front scalar value for the slab corresponding to the back scalar value of the previous slab can be held in a register and does not have to be retrieved again, there is no measurable performance benefit as the second lookup typically can be served from the texture cache.

On the other hand, the single-pass ray casting approach has several advantages when compared to slice-based volume rendering. As has been shown in Section 4.7.2, for a given sampling rate image fidelity of ray casting results is typically much higher than for slice-based rendering. This can be attributed to the full 32-bit float precision volume rendering pipeline and the equidistant sampling, even for perspective projections with unfavorable viewing conditions, realized by a ray casting shader. However, the best measure of volume rendering quality is the effectiveness and expressiveness of the visualization, which is primarily a matter of finding a suitable transfer function. In this respect, image fidelity is often secondary since minor rendering artifacts are often tolerated.

The biggest advantage, however, is the simplicity and flexibility of the single-pass ray casting approach. As has been demonstrated, acceleration techniques, such as early ray termination, empty-space leaping, and adaptive sampling, can be integrated with very little additional effort. Although such optimizations have been also described for slice-based volume rendering and multi-pass ray casting, these implementations lack the simplicity and elegance of the ray casting approach and require considerable more effort and implementation overhead. Potentially having the complete sampling history along the ray available and the explicit control over sampling enables many new possibilities. Furthermore, it is also very easy to realize arbitrary, nonstandard compositing schemes, like local MIP [183] or the even more complex Kubelka-Munk model discussed in Section 4.4.3, which go beyond OpenGL's alpha blending possibilities.

Another advantage of GPU-based ray casting, which is of interest at least for the near future, is the consistent use of 32-bit floating point precision throughout the volume rendering pipeline. Although high-accuracy 32-bit floating point blending is supported by current commodity graphics processors it is still very

costly on some GPUs. As has been shown in the previous section, the ray casting approach is up to 400% faster in some case. Furthermore, such high-accuracy blending is only available on the latest generations of GPUs. On older hardware it has to be emulated by ping-pong rendering at the cost of additional graphics memory consumption and increased memory bandwidth as well as higher complexity of the implementation. GPU-based ray casting, however, fulfills all precision requirements and at the moment can be even faster than slice-based rendering with 32-bit precision floating point blending.

In the end it should be noted that neither ray casting nor texture slicing is the better volume rendering approach in the first place. Both have their advantages and disadvantages. In general, what can be realized using ray casting can be also achieved by texture slicing and vice versa. However, it is often much simpler and requires considerably less effort to modify the ray casting approach in order to increase quality or performance than it is for a texture-slicing technique.

Chapter 5

GPU-based Isosurface Reconstruction

Isosurfaces are still the most prominent representation for visualizing volumetric scalar data sets. Isosurface rendering is a standard technique and, therefore, well understood by practitioners in a wide range of application areas. For this simple reason it is widespread used in medicine, computational fluid dynamics, or general engineering applications. In Chapter 4 already a method has been devised that allows the GPU-based visualization of isosurfaces. However, this method is restricted to uniform grids and can not directly deal with the unstructured grids that are very popular in many practical applications—computational fluid dynamics and finite-element analysis being only the most prominent examples. Furthermore, this approach visualizes the isosurface without actually reconstructing it. Therefore, it can be called an isosurface visualization technique in contrast to an isosurface extraction technique that yields an actual geometric representation of the surface. However, there are many applications that can benefit from the availability of an actual geometric representation or are possible only if such a polygonal representation of the isosurface is available. Having a polygonal representation available allows all kinds of postprocessing to be applied to the surface, including surface refinement and surface fairing techniques. Furthermore, the whole lot of standard computer graphics techniques devised for polygon meshes can be directly applied, such as complex shading and shadowing methods. Another important point is computational efficiency. The techniques presented in Chapter 4 require to recompute the isosurface whenever view parameters or surface properties are changed. A polygonal representation of the surface, however, can be repeatedly rendered much more efficiently once it is available.

That said, the most common isosurface visualization methods can be roughly separated into two distinct classes. The first are geometric methods that fit a polygonal approximation to the level set of the scalar field. For structured grids

the *Marching Cubes* algorithm by Lorensen and Cline [132] solves this problem. Due to its algorithmic simplicity and generality Marching Cubes is still the most widely used algorithm for isosurface visualization and available in most commercial visualization systems. Numerous variants and enhancements of the original algorithm have been proposed; including hierarchical approaches using spatial [227] as well as data domain subdivisions [131] to reduce the number of grid cells that are visited during construction of the surface, and methods that guarantee the topological correctness of the extracted surface [126]. Furthermore, it has been adopted for unstructured grids, i.e. tetrahedral grids, as well [41]. The so-called *Marching Tetrahedra* algorithm will be discussed in detail in Section 5.1. On the other hand, due to the huge amount geometric primitives that have to be created and transfered to the GPU for rendering, for reasonably sized grids these techniques do usually not allow visualization and extraction at interactive rates.

On the other side there are GPU-assisted approaches that create visualizations of isosurfaces without actually reconstructing a geometric representation but at interactive rates. Most of this work concentrates on uniform grids. Westermann and Ertl [221] employ the alpha test to render isosurfaces with a slice-based volume renderer. In their approach shading is realized using the color matrix transform provided by the OpenGL imaging subset [221] or in a much more general fashion by using a register combiner setup [216]. By choosing an appropriate pre-integration table, pre-integrated direct volume rendering can be also applied for isosurface rendering as has been demonstrated by Engel et al. [50]. In addition,there are also forward-projection methods based on the splatting approach [242] and point-based isosurface rendering methods [29]. A GPU-based ray casting approach for uniform grids has been described in Chapter 4.

GPU-assisted approaches have been also devised for unstructured tetrahedral grids. Röttger et al. [178] and Weiler et al. [217] have used cell-projection [178] to render multiple shaded isosurface of an unstructured mesh. A GPU-based ray casting approach was proposed by Weiler et al. [218].

Lately also considerable effort has been spend on methods for parallelizing the Marching Tetrahedra and Marching Cubes algorithm using the graphics processing unit. Pascucci [157] and Reck et al. [168] were the first to used the vertex processing units to extract an isosurface from an unstructured tetrahedral grid. Both algorithms are very similar. For each tetrahedron a single quadrilateral is rendered such that its vertices are provided with additional information that is required for calculating the intersection of a tetrahedron with the isosurface. Once an intersection has been found, the quadrilateral's vertices are moved to the corners of the intersection polygon. Since each vertex is processed independently by the vertex shader this involves a large overhead because the complete data characterizing the cell has to be send for each of the four vertices of the quadrilateral and the classification and interpolation for all possible intersection points has to be re-

peated for each vertex of the quad. Buatois et al. [23] improved on this approach by storing the grid in texture objects instead of downloading the whole data for each rendered frame. Johansson and Carr [89] precompute cell topology for each cell of a regular grid and use display lists for each of the Marching Cubes cases to speed up the rendering.

However, all of these approaches do not generate geometry. Like the pixel-based methods mentioned before, they only create a visual representation of the surface by rendering an appropriate number of triangles for each cell intersected by the isosurface. Further, they cannot harness the full computational power of the GPU since they operate on the compared to the number of fragment processors less abundantly designed vertex processing units. Thus, these solutions have both limitations regarding flexibility and performance.

In this chapter an approach will be presented that overcomes this limitations by storing the polygonal isosurface geometry extracted from data given on an unstructured tetrahedral grid in GPU memory. This technique, first published in collaboration with Simon Stegmaier [102], stores the complete grid data in texture memory, thus avoiding costly bus transfers, and utilizes the much more powerful fragment processing units to generate the isosurface polygons. Since the geometrical reconstruction of the surfaces is stored in an on-board graphics memory buffer object that can be alternatively bound as a render target or a vertex array, no expensive framebuffer readback to system memory is required for repeatedly rendering the surface or applying subsequent GPU-based postprocessing steps, such as for example Laplacian smoothing of the mesh geometry.

Kipfer and Westermann [98] improved upon this method by introducing an edge-based classification scheme that both reduces the size of the necessary data structures as well as the number of memory accesses and edge intersection computations.

5.1 Marching Tetrahedra Revisited

As mentioned above, the Marching Tetrahedra algorithm by Doi and Koide [41] in principle is a simple variation of the original Marching Cubes [132] idea. Since it provides the basis for the approach presented in the following, in this section a brief overview of the basic principle of the algorithm will be given.

Due to the simple topology of the linear interpolant inside the tetrahedral cell, there are—by exploiting simple symmetries—only three distinct configurations possible for the isosurface inside the cell. These are shown in Figure 5.1. There are at most four intersections of the isosurface with the tetrahedron's edges. Furthermore, since the interpolated data values also vary linearly in the interior of the cell these surfaces are planar, i.e. only a single simple polygon per cell,

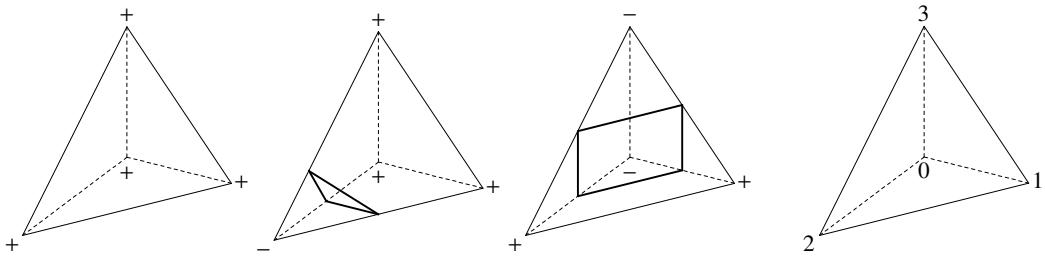


Figure 5.1: The three basic cases of the Marching Tetrahedra algorithm and the local vertex ordering of a tetrahedral cell. An isosurface inside a linear tetrahedron is either empty, a triangle, or a quadrilateral.

and, accordingly, the algorithm does not suffer from ambiguities like Marching Cubes [44, 235]. In other words, the isosurface inside a linear tetrahedron is either empty or can be represented by a triangle or a planar quadrilateral in the nontrivial cases.

Similar to Marching Cubes the basic algorithm processes one grid cell after the other. First the cell is classified by assigning a positive value (or 1) to vertices where the corresponding scalar value is larger or equal to the specified isovalue and a negative value (or 0) to vertices where the corresponding value is less than the isovalue, resulting in a 4-bit classification mask for the tetrahedron. This classification mask in turn can now be used to look up in a table the corresponding set of edges cut by the isosurface and the ordering of the intersection points necessary for defining the isosurface polygon. The actual intersection positions can then be determined by linear interpolation between the vertices constituting the individual edges.

In the following it will be shown, how this algorithm can be implemented in a parallel fashion using the GPU's fragment processing units. Conceptually it is split into two or more passes. First, the extraction pass that generates the isosurface polygons by classifying each tetrahedron in the grid, finding the intersection positions, and storing them into a graphics memory buffer object. Second, the postprocessing or rendering passes that use the previously filled buffer objects as input bound to a vertex array. In addition, since at the time of development the functionality of GPUs was rather limited compared to today some adjustments of the original algorithm are suggested that reduce the number of memory access operations and arithmetic instructions of the shader to a minimum. Although the progress in GPU technology has alleviated most of the restrictions of then, the proposed solutions may be still of interest since memory access is an increasing bottleneck in todays highly clocked graphics processor cores. In order to hide the latency of texture lookups high-arithmetic density is beneficial, i.e. a low texture instructions to arithmetic instructions ratio is recommended.

5.2 Data Encoding

Extracting isosurface geometry on the GPU requires the input data as well as the resulting output geometry to be suitably encoded into data structures accessible to the processing units of the GPU. In case of the fragment processor these are textures and render buffers. In the following two sections the data structures used to encode the unstructured input grid and the output polygons in textures and off-screen render buffers will be discussed.

5.2.1 Input Encoding

Since the input data is given on an unstructured tetrahedral grid, not only the scalar data values but also vertex positions and the grid connectivity have to be stored appropriately in texture objects. Using an indexed encoding scheme and assuming the grid consists of M tetrahedral cells that share N vertices and further assuming vertex coordinates as well as data values are stored in single precision floating point format the minimal theoretical storage requirements are $128N + 4M\lceil \log_2 N \rceil$ bit. However, for an efficient GPU implementation the number of vertex index bits cannot be freely chosen, since available texture formats either store 8, 16, or 32 bits per component. Using a 24-bit index stored in a 8-bit RGB texture—in the following referred to as the *index texture*—proved to be a good compromise between the storage required for the index texture and the total number of vertices that can be addressed. Those split indices can be used straightforward to address the texels of a 3D RGBA texture object—the *vertex texture*. Theoretically this allows to store up to 16 million vertices and their corresponding scalar values in a single texture object that, again assuming 32 bits to store each vertex components and the corresponding scalar values, amounts to 256MB of total storage for vertex data which in turn equals the total amount of texture memory available on ATI R300-based graphics boards. However, in practice the available texture memory has to be shared with the index texture, the framebuffer, the output buffers, and additional graphics buffers, storing for example executable shader code. Furthermore the employed hardware does not support non-power-of-two textures, thus the maximum size of a single vertex texture is $256 \times 256 \times 128$ which limits the number of vertices that can be effectively stored in a single vertex texture to approximately 8.3 million.

In order to reduce the number of expensive memory accesses and since, as will be elaborated in Section 5.3, the hardware used to implement the algorithm enforces rather strict limits on the number of texture lookups, in particular on the number of texture indirections, an additional 32-bit float 2D texture is used that stores the scalar values for each tetrahedron separately. One four component RGBA texel is used per tetrahedron. This texture will be called *scalar texture* in

the following. In total this amounts to 128 bits of per-vertex data and additional 224 bits per tetrahedron.

Compared to the vertex shader centric approaches proposed by Pascucci [157] and Reck et al. [168] that allow streaming of the data, using texture objects for storing the input data suffers a considerable drawback. If the data set is too large to fit into graphics memory it has to be split into a set of textures that have to be loaded consecutively from main memory. Furthermore it is also not easy to employ typical optimization strategies like using spatial or data range decompositions in order to process only those tetrahedra that are at least likely to be intersected by the isosurface. Therefore, a partitioning strategy that aims for optimal texture memory utilization and minimizes overlap between the texture segments has been investigated in [147].

5.2.2 Output Encoding

The major problem in generating geometry using the fragment processing units is the necessity to redefine the semantics of the output pixels in order to reinterpret the pixels' color values as vertex data that can be re-injected into the vertex pipeline. While since the introduction of 32 bit floating point render targets there is no problem to store high-precision floating point data produced by the fragment shader in a render buffer and using this data in subsequent rendering pass as texture input to the fragment shader, there is no such path in standard OpenGL that allows to use rendered data as input streams to vertex processing without expensively copying the data.

Basically, there are two problems that normally prevent the use of a render target as a vertex input stream. Although both objects are represented by chunks of graphics memory there is a profound difference in the organization of the data elements. In other words, the internal data organization is bound to and optimized for the semantics of the data. While textures, and accordingly also render targets are organized as 2D arrays that are typically stored in a cache-friendly swizzeled format optimized for spatially close access in 2D texture space, vertex data is stored in linear one-dimensional arrays, according to the typical GPU access patterns to the data.

The experimental ATI *SuperBuffers* OpenGL extension [137] offers a solution to this problems. It provides a unified GPU memory model based on generic graphics memory objects that, in contrast to textures and vertex buffer objects, have the semantics of the data separated from the actual storage. That is, it provides the means to allocate GPU memory objects that can be transparently bound as a render target, i.e. pixels, texture object, i.e. texels, and vertex arrays, i.e. vertex attributes stream. This provides a lightweight possibility to generate geometry on the GPU by re-injecting render target content as vertex input to the graphics

pipeline without ever having to read back any data to main memory.

In the following the organization of the output vertex data is described. As discussed in Section 5.1, there are at most four intersections of the isosurface with a tetrahedron's edges. Either there is no intersection of the surface and the tetrahedron, i.e. there is no polygon at all, or there are three or four intersections resulting in a triangular or a quadrilateral polygon. Thus up to four three-dimensional vectors per tetrahedron are needed to store the vertices of the resulting polygons. However, all three cases can be handled uniformly because OpenGL allows the rendering of degenerated polygons, i.e. the vertices of a polygon are allowed to coincide. Thus, a triangle can be represented by replicating one of the four vertices of a quadrilateral. Accordingly, a quad primitive with four identical vertices is discarded altogether during rasterization. Therefore, the output buffer is realized using a SuperBuffer object that is bound as a 32-bit floating point RGBA render target. Each consecutive group of four pixels in a horizontal scanline is associated with the four vertices of a possibly degenerated intersection polygon respectively the intersected edges of the tetrahedron corresponding to this intersection. This conceptual organization in groups of four 4D vector elements directly corresponds to a tightly packed vertex array holding OpenGL quads.

For a given render target of $W \times H$ pixels in size this results in $H \times W/4$ four pixel groups, i.e. the maximum render buffer size of 2048^2 allows to store the isosurface intersection polygons of more than one million tetrahedral cell in a single SuperBuffer object. Accordingly, in a single extraction pass more than a million tetrahedra can be processed in parallel. Conversely, if the number of tetrahedral cells potentially intersected by the isosurface exceeds $H_{\max} \times W_{\max}/4$ they must be divided in several chunks processed in sequence.

5.3 Extraction Algorithm

This section will elaborate on the actual realization of the extraction algorithm in a fragment shader program. Roughly speaking the idea is as follows. First, the appropriate vertex, scalar, and index textures, as described in Section 5.2.1, are bound and a render buffer according to the previous section is set up. Then, the actual computation of the isosurface geometry is initiated by rendering a polygon that covers all pixels of the output buffer and triggers the execution of a fragment shader program that implements the extraction algorithm as will be detailed in the following. Figure 5.2 gives a conceptual overview of the extraction algorithm for a single edge intersection.

According to the output data layout described in Section 5.2.2 always four fragments generated consecutively in a horizontal scanline correspond to the edge intersections of a unique tetrahedron. This in turn defines a mapping from frag-

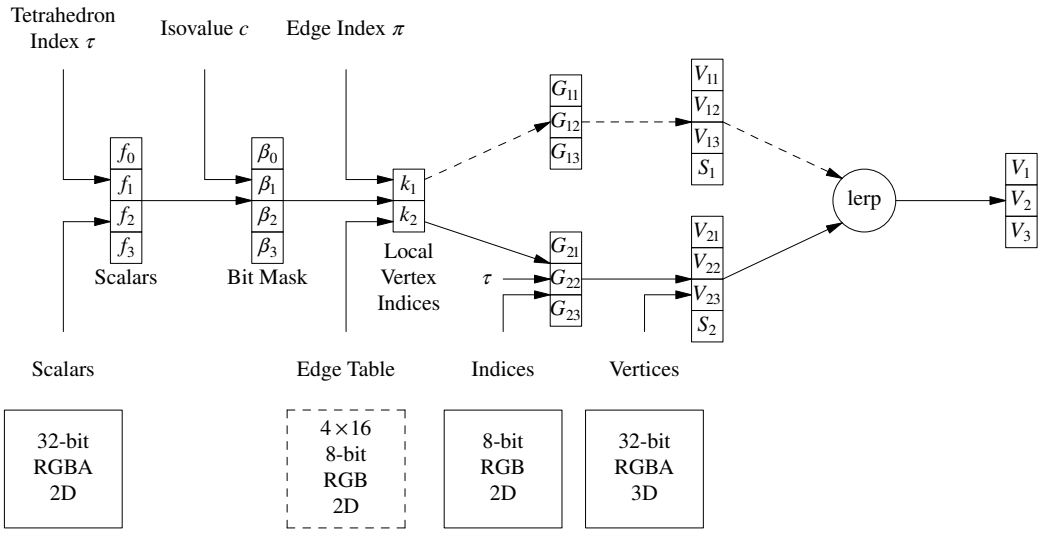


Figure 5.2: Conceptual overview of the extraction algorithm. The computation of a single edge intersection, i.e. the computations done in a single fragment shader invocation is shown.

ment position, i.e. window coordinates, (x_f, y_f) to tetrahedra. Thus, for a given fragment the unique index τ of the corresponding tetrahedron can be computed as

$$\tau = \begin{pmatrix} \lfloor \frac{x_f}{4} \rfloor \\ y_f \end{pmatrix}.$$

Let $k = 0, \ldots, 3$ denote the local index of the vertices within the tetrahedron. Given the tetrahedron's index τ the corresponding four-component vector of scalar values $\mathbf{F} = (f_k)^T$ can be retrieved by sampling the previously described scalar texture. Since texture sampling operations require coordinate values to be in the range $[0, 1)$, for the actual implementation τ has to be scaled appropriately. In the following, however, this implementation detail will be neglected for the sake of simplicity and clarity and only integral index representations will be used. Now it becomes also clear, why an additional 2D texture map storing only the scalar values is used. This way, classifying the whole tetrahedron can be based on a single four element lookup in a 2D texture instead of four 2D lookups in order to acquire the indices followed by another four lookups to retrieve the scalar values from the vertex texture.

Given the user-defined isovalue c, a bit mask

$$\beta = (\beta_k)^T, \quad k = 0, \ldots, 3 \quad \text{with} \quad \beta_k = \begin{cases} 0 & \text{if } f_k < c \\ 1 & \text{else} \end{cases}$$

is computed that classifies the vertices of the tetrahedron as either *inside* (0) or

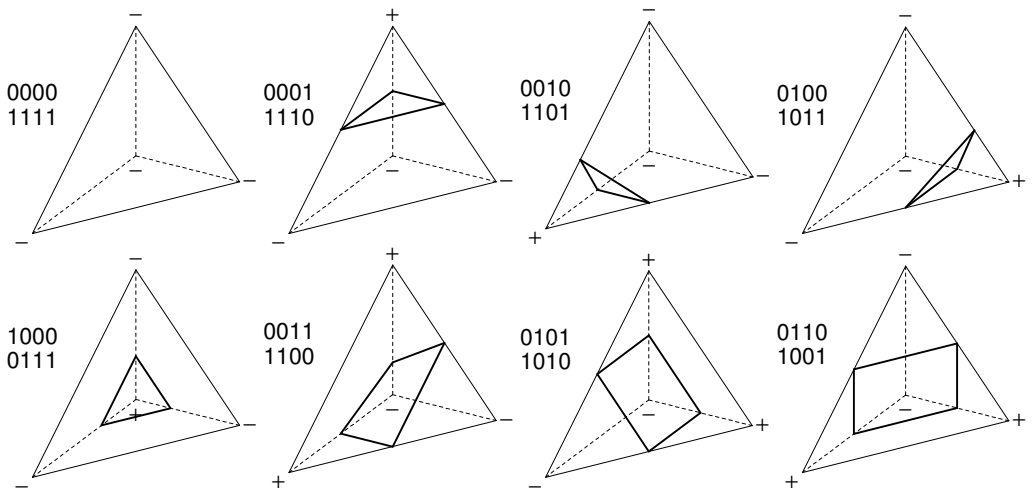


Figure 5.3: The eight cases of the Marching Tetrahedra algorithm.

outside (1) the isosurface depending on whether the associated scalar value is less than or greater than the isovalue. Given the unique local order of the vertices of a tetrahedron β also determines which edges of the tetrahedron are actually intersected by the isosurface. Figure 5.3 shows the sixteen cases that have to be taken into account. Note, that each of the shown cases represent a symmetric pair of classifications, i.e. the same edges of the tetrahedron with respect to the local ordering of vertices as shown Figure 5.1 are intersected. Only the signs of the vertices with respect to the isovalue has changed. In other words, the two cases represented by a bit mask and its bitwise complement characterize a pair of triangles that share the same vertices but require different winding orders. Hence, these symmetries can only be exploited if the actual orientation of the polygons is irrelevant. However, the winding order is important for various lighting calculations and polygon culling.

Since all intersections between isosurface polygons and tetrahedra edges are computed in parallel an additional index—the *edge index* $\pi = x_f \bmod 4$—is used to decide which potential intersection has to be computed for the actual fragment.

Using the decimal interpretation of β and the edge index π it is possible to look up the correct local vertex indices k_1 and k_2 of the respective edge end points for each fragment in a table. These two vertex indices combined with the tetrahedron index τ, in turn, can be used to look up the actual global vertex indices $\mathbf{G}_1$ and $\mathbf{G}_2$ in the index texture. Using τ as the tetrahedron's base address and the local vertex indices as offsets the index for this lookup is computed as

$$\mathbf{r}_i = \begin{pmatrix} 4\tau_1 + k_i \\ \tau_2 \end{pmatrix}, \qquad i = 1, 2.$$

The three-component vertex indices $\mathbf{G}_1$ and $\mathbf{G}_2$, again, provide the coordinates for accessing the 3D vertex texture that yields the actual vertex coordinates $\mathbf{V}_1$ and $\mathbf{V}_2$ of the edge end points. Linear interpolation

$$\mathbf{V} = t\mathbf{V}_1 + (1-t)\mathbf{V}_2, \quad t = \frac{c - f_{k_1}}{f_{k_2} - f_{k_1}}$$

finally yields the actual edge intersection coordinates that are written in the output buffer. Now it becomes clear why a RGBA texture instead of a RGB texture is used for storing the vertex coordinates: As fragment programs currently do not support the concept of indirect addressing, it would be very complicated in terms of fragment program instructions to select the right scalar values f_{k_1} and f_{k_2} from the scalar vector $\mathbf{F}$. Therefore, the scalar values is stored redundantly in the alpha component of the vertex coordinates texture and the in scalars texture. The interpolation weight t can then be directly computed from the fourth components of the two vertices. However, in terms of texture memory consumption, this is no overhead, since RGB textures are internally mapped to RGBA textures, anyway.

Unfortunately, the algorithm as described above cannot be implemented on the target hardware. It contains too many texture indirections, i.e. texture lookups depending on source coordinates that have been computed from the result of a previous texture sampling operation. The native limit on texture indirections on the Radeon 9800 GPU is four; which the naïve solution exceeds by two. There are two possibilities to reduce the length of the texture dependency chain and to circumvent this limitation. First, one could give up the indexed encoding scheme in favor of direct encoding of the grid, i.e. replicating the vertex and scalar data for each tetrahedron. However, this would result in a drastic increase of memory consumption. The other—and only reasonable—solution is to emulate the edge table lookup by arithmetic operations. How this can be realized will be outlined in the following section.

5.3.1 Emulating the Edge Table

Replacing the edge table lookup by arithmetic instructions is not easy and has to take the idiosyncrasies of the GPU's instruction set into account.

In order to simplify the computations we start by reducing the number of cases to the eight symmetric cases shown in Figure 5.3 by inverting β if necessary. This leaves the edge table as shown in Table 5.1. The decision whether it is necessary to invert the bit mask can be based on the following observations. First, all bit masks that must be inverted, when interpreted as a binary representation, represent a decimal value that is larger or equal to seven. Furthermore, adding the number of bits set in β to its decimal representation yields a value larger than nine in all cases the mask has to be inverted, while it is less than or equal to nine for all others.

Table 5.1: Edge table for the lookup of local vertex indices k_1 and k_2 that constitute the intersected edges depending on tetrahedron classification bit mask β and edge index π. Symmetric cases shown in parentheses.

	π				
β^T	0	1	2	3	
$(0,0,0,0)_{/(1,1,1,1)}$	$\{0,0\}$	$\{0,0\}$	$\{0,0\}$	$\{0,0\}$	empty
$(0,0,0,1)_{/(1,1,1,0)}$	$\{3,0\}$	$\{3,1\}$	$\{3,2\}$	$\{3,2\}$	
$(0,0,1,0)_{/(1,1,0,1)}$	$\{2,3\}$	$\{2,0\}$	$\{2,1\}$	$\{2,1\}$	triangle
$(0,1,0,0)_{/(1,0,1,1)}$	$\{1,2\}$	$\{1,3\}$	$\{1,0\}$	$\{1,0\}$	
$(1,0,0,0)_{/(0,1,1,1)}$	$\{0,1\}$	$\{0,2\}$	$\{0,3\}$	$\{0,3\}$	
$(0,0,1,1)_{/(1,1,0,0)}$	$\{0,2\}$	$\{0,3\}$	$\{1,3\}$	$\{1,2\}$	
$(0,1,0,1)_{/(1,0,1,0)}$	$\{0,1\}$	$\{0,3\}$	$\{2,3\}$	$\{2,1\}$	quadrilateral
$(0,1,1,0)_{/(1,0,0,1)}$	$\{1,0\}$	$\{1,3\}$	$\{2,3\}$	$\{2,0\}$	

Therefore, the simple criterion $\beta \cdot (9,5,3,2)^T > 9$ is sufficient to distinguish the cases.

Next, the edge table can be divided into the three major cases that will be treated separately in the following. For each case a sequence of shader instructions has to be found that computes the correct local vertex index pair that constitute the intersected edge $\{k_1, k_2\}$ depending on the bit mask β and the edge index π. Besides, there are two additional objectives. First, vertex replication for the triangular and the empty case has to be handled correctly and, second, the number of fragment shader instructions necessary for the computation should be as low as possible.

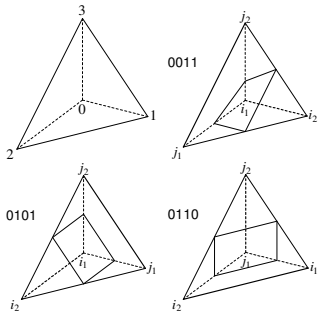
The Triangle Case For the triangle case the following observation can be made. All intersected edges share a single common vertex and the local index of this vertex is defined by the single bit set in the classification mask β. Hence, the first index can be compute by the simple dot product

$$k_1^{\triangle} = \beta \cdot \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix},$$

regardless of the actual edge index β. Computing the second index is also not very complicated. Given the value of $k_1^{\triangle}$ it is sufficient to enumerate the remaining

Table 5.2: Reduced edge table for the three quadrilateral cases shown on the right.

	π			
β^T	0	1	2	3
	$\{i_1, j_1\}$	$\{i_1, j_2\}$	$\{i_2, j_2\}$	$\{i_2, j_1\}$
$(0,0,1,1)$	$\{0,2\}$	$\{0,3\}$	$\{1,3\}$	$\{1,2\}$
$(0,1,0,1)$	$\{0,1\}$	$\{0,3\}$	$\{2,3\}$	$\{2,1\}$
$(0,1,1,0)$	$\{1,0\}$	$\{1,3\}$	$\{2,3\}$	$\{2,0\}$



three indices in the correct order. And, second, to replicate one of those indices in order to create a quad that is degenerated to a triangle. This can be accomplished, e.g., by the following rule:

$$k_2^{\triangle} = (k_1^{\triangle} + 2\max(0, \min(1, \frac{\pi}{2})) + 1) \bmod 4.$$

It correctly replicates the last index, i.e. yields the same result for $\pi = 2$ and $\pi = 3$, and, fortunately, maps to only a total of six shader assembly instructions. Each pair of local edge indices of the four triangle cases shown in Table 5.1 can therefore be computed with only seven native ALU operations.

The Quadrilateral Case Finding a similar solution for the quadrilateral case is slightly more complex. Although in total there are only three cases, the ordering of local indices is much more complicated. Lets start with a few observations. First, the isosurface intersects four of six edges of the tetrahedron and all of its four faces. That means, there are only two edges left, lets say $\{i_1, i_2\}$ and $\{j_1, j_2\}$, that are not intersected. Second, these two edges cannot share a common vertex. If this would be the case, these edges would span a face that is not intersected by the isosurface. This is not possible. From that follows that all intersected edges can be found by connecting the end points of the edges that are not intersected. In other words, the intersected edges are given by elements of the Cartesian product $\{i_1, i_2\} \times \{j_1, j_2\}$. Third, the set of the end points of the not intersected edges corresponds to all vertices of the tetrahedron. That means one vertex index can be chosen freely. In fact, the same index can be chosen for all three quadrilateral cases. Thus, e.g., choosing $j_2 = 3$ for all three cases fixes also j_1 for all cases. See right images in Table 5.2. However, there is another degree of freedom in choosing i_1 and i_2. Demanding the same index pair to be chosen for an edge intersection regardless of the specific case, leads to the index ordering shown in the

reduced edge table given in Table 5.2. In other words, for each classification vector β there is a unique choice of indices $\{i_1, i_2, j_1, j_2\}$ that can be used to describe all intersected edges.

Given the fixed choice of j_2 we need a way to compute the remaining three indices. Defining all indices as dot products of the classification bit mask β with unknown but constant vectors $\mathbf{I}_1$, $\mathbf{I}_2$, $\mathbf{J}_1$, and $\mathbf{J}_2$, i.e.

$$i_1 := \beta \cdot \mathbf{I}_1, \quad i_2 := \beta \cdot \mathbf{I}_2, \quad j_1 := \beta \cdot \mathbf{J}_1, \quad j_2 := \beta \cdot \mathbf{J}_2,$$

three systems of linear equations can be derive based on the content of the columns of the reduced edge table. For example, solving for $\mathbf{I}_2$ based on the condition $i_2 = \beta \cdot \mathbf{I}_2$ leads to the following system of equations

$$\begin{array}{ccccccc} & & I_{23} & + & I_{24} & = & 1 \\ I_{22} & & & + & I_{24} & = & 2 \\ I_{22} & + & I_{23} & & & = & 2 \end{array}$$

with solution $\mathbf{I}_2 = (0, 3/2, 1/2, 1/2)^T$. Analogous computations yield the vectors $\mathbf{I}_1$, $\mathbf{J}_1$, and $\mathbf{J}_2$ respectively. Thus, all indices that constitute the not intersected edges $\{i_1, i_2\}$ and $\{j_1, j_2\}$ can be computed very efficiently by the following four dot products.

$$i_1 = \beta \cdot \mathbf{I}_1 = \frac{1}{2}\,\beta \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \\ -1 \end{pmatrix}, \quad i_2 = \beta \cdot \mathbf{I}_2 = \frac{1}{2}\,\beta \cdot \begin{pmatrix} 0 \\ 3 \\ 1 \\ 1 \end{pmatrix},$$

$$j_1 = \beta \cdot \mathbf{J}_1 = \frac{1}{2}\,\beta \cdot \begin{pmatrix} 0 \\ -1 \\ 1 \\ 3 \end{pmatrix}, \quad j_2 = \beta \cdot \mathbf{J}_2 = \frac{1}{2}\,\beta \cdot \begin{pmatrix} 0 \\ 3 \\ 3 \\ 3 \end{pmatrix}.$$

However, there is still the problem of finding the correct combination of indices for a given edge index π. As given in Table 5.2 the order is $[i_1, i_1, i_2, i_2]$ for the first index and $[j_1, j_2, j_2, j_1]$ for the second index. However, this can be achieved by modifying the vectors in the previously derived dot products. From the equalities

$$\mathbf{I}_2 = \mathbf{I}_1 + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{and} \quad \mathbf{J}_2 = \mathbf{J}_1 + \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix},$$

the following two simple formulas can be derived

$$k_1^{\square} = \beta \cdot \left[\frac{1}{2} \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} + \left\lfloor \frac{\pi}{2} \right\rfloor \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right]$$

$$k_2^{\square} = \beta \cdot \left[\frac{1}{2} \begin{pmatrix} -1 \\ 1 \\ -3 \end{pmatrix} + \pi(3-\pi) \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} \right]$$

that correctly compute the local index pairs according to Table 5.2 given the classification mask β and the edge index π. This is slightly different to the bit-mask–based selection method proposed in the original publication [102]. However, this solution is much more efficient and reduces the overall instruction count of the shaders assembler source by ten instructions.

Case Selection and the Empty Case Selecting the correct case, i.e. determining whether the index pair $(k_1^{\triangle}, k_2^{\triangle})$ or the index pair $(k_1^{\square}, k_2^{\square})$ has to be computed, can be accomplished by counting the number $b = \beta \cdot (1,1,1,1)^T$ of bits set in β. If this product equals two, the result is a quadrilateral. Otherwise, there was at most one bit set in β which indicates the triangle case or no intersection. Unfortunately, the ATI R300 GPU does not support dynamic flow control in the fragment shader. Therefore both cases have to be evaluated and the correct pair has to be selected.

Last, we have to account for the empty case. However, this is trivial. Since b equals zero for the empty case and is larger or equal in all other case, it is sufficient to multiply k_1 and k_2 by a factor of $\max(\min(b,1),0)$ which equals zero in the empty case and one for all other cases. Thus, the four vertices will be correctly replicated. However, it is beneficial to move this multiplication to the end of the algorithm and to multiply the components of the interpolated intersection vertex V instead. This results in a all-zero vertex position that can be easily culled during frustum clipping.

5.4 Rendering and Postprocessing

The simplest form of postprocessing that can be applied to the extracted isosurface polygons is rendering the surface. This is straightforward since the output render buffer can be directly bound as a vertex array buffer. However, in addition to the vertices stored in the buffer normal vectors are needed for shading the surface. The simplest type of shading—flat shading using a constant normal vector per triangle—can be accomplished by using precomputed gradient information that is stored per tetrahedron in an additional texture map similar to the index

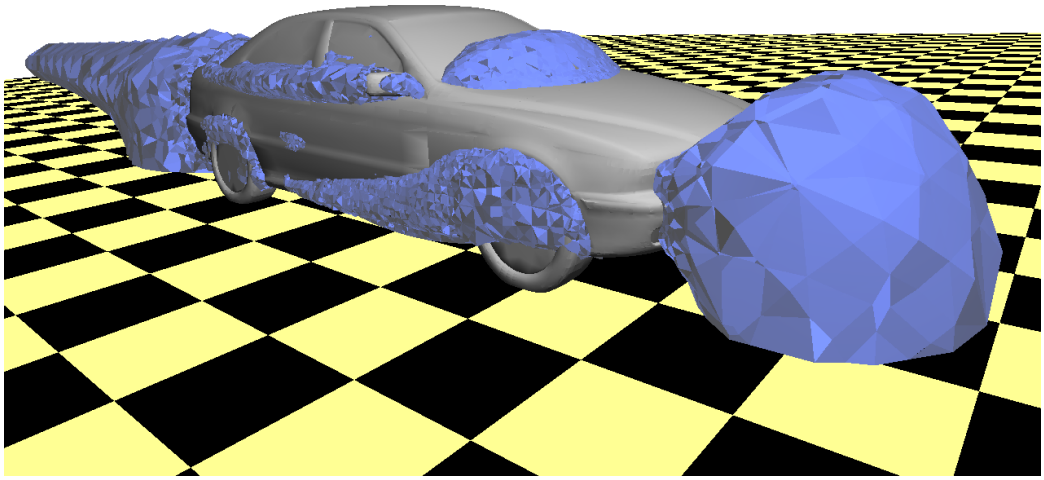


Figure 5.4: Isosurface geometry extracted from an unstructured data set[I] on the GPU. An isosurface of the velocity magnitude of an air flow around the car body is shown.

texture. When rendering the surface this texture is accessed using the global tetrahedron index encoded in the alpha channel respectively the w-component of the computed vertices in the extraction pass.

More sophisticated postprocessing, e.g. rendering smooth-shaded surfaces or Laplacian-smoothing, however, requires additional information about the connectivity of the surface mesh. Since the result of the extraction algorithm described in the previous section is nothing more than a polygon soup, i.e. a list of vertices that describes a quad mesh inclusive of polygons that are degenerated to triangles or points, additional adjacency information must be provided that allows to access vertex data of neighboring polygons. A solution that uses precomputed adjacency tables is described by Mohr in his diploma thesis [147].

5.5 Evaluation

Figure 5.4 shows an example of an isosurface extracted and visualized with the described GPU-based Marching Tetrahedra algorithm. The image shows a visualization of the velocity magnitude field extracted from a CFD data set of the flow

[I]Dataset courtesy BMW AG.

around a car body. The data set consists of 448.450 tetrahedral cells. Extracting the isosurface polygons took approximately 37ms on an ATI Radeon 9800Pro graphics board featuring 256MB of graphics memory. Once extracted, rendering the resulting surface polygons from the SuperBuffer is possible at a rate of 46 frames per second.

In general extraction rates of approximately 12 million tetrahedra per second can be achieved on this GPU. Table 5.3 shows some results for a number of data sets of various sizes, including both typical benchmark data sets for unstructured grids, an example from CFD simulation in automotive engineering, and Cartesian data sets that were decomposed into tetrahedra in a preprocessing step.

Of course this extraction rate cannot be sustained when the necessary data does not fit into GPU memory and textures have to be swapped to and from main memory. Therefore, only a hybrid solution that exploits preselection of cells based on the isovalue range on the CPU and computes the isosurface polygons in parallel on the GPU will be able to deliver high extractions rates for very large data sets. However, such optimization techniques cannot be employed when the visualization data is postprocessed using the GPU or is created on the GPU on the first hand.

Nevertheless, even if such preselection techniques are used to reduce the number of tetrahedra that have to be processed, the large memory footprint necessary for storing the data structures remains the major problem. Especially the size of the SuperBuffer objects is the limiting factor. While part of the input textures can be re-used for successive extraction passes, the output must be kept in memory in order to to take advantages of the extracted geometry. Given the memory layout described in Section 5.2 the overall GPU memory required in bytes is

$$S_{\text{total}} = 4s2^{\max(1,\lceil \log_2 \frac{N}{256^2} \rceil)} + H_{\max} \cdot W_{\max}/4 \left(4s + (4+4s) \left\lceil \frac{M}{H_{\max} \cdot W_{\max}/4} \right\rceil_{W_{\max}} \right)$$

where s is the number of bytes necessary to store a single component of the input vertex coordinates and of the output positions respectively. Taking framebuffer memory and other small memory buffers, for example shader objects, into account and assuming 32-bit floating point values, i.e. $s = 4$, approximately 2.1 million isosurface polygons, i.e. tetrahedron intersection, can be extracted to on-board memory buffers given the 256MB of graphics memory. However, in many cases the accuracy of the stored vertex positions can be reduced. In the majority of cases a 16-bit floating point format provides sufficient accuracy to represent the vertices of the isosurface but more than doubles the number of isosurface polygons that can be stored in GPU memory to more than 5.2 million. Additional reduction of the memory footprint might be achieved using the stream compactification techniques described by Dyken et al. [45].

Table 5.3: Properties of various evaluation data sets and corresponding extraction and rendering performance.

Data set	# Vertices	# Tetrahedra	# Passes	Tetrahedra/s Extraction	Tetrahedra/s + Rendering
Tornado	1,000,000	4,851,459	5	11.76M	5.51M
Neghip	262,144	1,250,235	2	11.97M	7.56M
Car	85,843	448,450	1	12.02M	7.62M
Bluntfin	40,960	224,874	1	12.02M	7.65M

Another problem may be the availability of functionality to render directly into a vertex buffer object. The SuperBuffer OpenGL extension never reached the level of a official OpenGL extension. There was never an official specification and the functionality was never officially available in publicly released drivers but was exposed only in experimental beta drivers by ATI. Unfortunately, in the meantime the proposal was withdraw in favor of the framebuffer objects extension. Sadly, this extension is not as general as the framework proposed in the SuperBuffer extension. In particular the possibility to directly bind a texture or render target as a vertex input stream has been skipped. An alternative solution has been provided by ATI by their Render-to-Vertex-Buffer extension for Direct3D 9 or more recently by the transform feedback extension.[II]

Besides this disadvantages there are also some positive points. First, compared to other hardware-based techniques that can handle unstructured grids there is no need for sorting the cells, as it is required by cell projection. Second, an actual geometric representation of the isosurface is available that facilitates repeatedly rendering the surface or applying subsequent GPU-based postprocessing.

It is also possible to extract isosurfaces from structured grids by decomposing the cells into tetrahedra. Of course, this increases both the number of grid cells that have to be processed as well as the number of triangles of the resulting isosurface compared to Marching Cubes. There are two possibilities to decompose a cubic cell into tetrahedra without adding additional vertices. A hexahedral cell can be decomposed into five or six tetrahedra. Both variants have advantages as well as disadvantages. Assuming the indexed data structures described in Section 5.2.1 the storage requirements for the connectivity information of the decomposition of a uniform grid is 20% higher in the latter case. The total storage requirements for a uniform grid of size $N_x \times N_y \times N_z$ is

[II] http://www.opengl.org/registry/specs/EXT/transform_feedback.txt

$N_x N_y N_z S_V + 5 \cdot 4(N_x - 1)(N_y - 1)(N_z - 1)S_I$ for a 5-tetrahedra decomposition and $N_x N_y N_z S_V + 6 \cdot 4 \cdot 4(N_x - 1)(N_y - 1)(N_z - 1)S_I$. Where S_V and S_I denote the storage required for a scalar data value and a vertex index respectively. Although this may impose the impression that the minimal 5-fold decomposition is superior compared to the 6-tetrahedra decomposition, there are also disadvantages. Such a decomposition is not even. The central tetrahedron has twice the volume compared to the remaining four which leads to notable artifacts. A detailed study of the artifacts caused by different tetrahedral subdivision schemes can be found in Carr et al. [26]. Only recently GPU-based Marching Cubes implementations that extract geometry have been presented by Dyken et al. [45].

Chapter 6

Point-based Quadric Glyphs

While for scalar and vector fields natural and descriptive metaphors, like semi-transparent volume rendering or isosurface representations respectively streamline visualizations, exist, there are no such direct physically motivated representations for higher-order and multi-variate data fields, like tensor field data. Typical approaches reduce the dimensionality of the problem by either computing derived scalar or vector valued quantities and visualizing this fields instead—using global or geometric methods—or by extracting and visualizing a suitably defined abstract feature representation. Such an abstract representation is often achieved by sampling or probing the field at appropriate positions and mapping its local attributes to various geometrical properties, such as shape, size, or orientation, and surface properties, e.g. color or texture, of application specific icons or glyphs.

The use of abstract iconic representations for the display of high-dimensional statistical data has a long tradition in the field of information visualization—Star glyphs [52], Kleiner-Hartigan trees [104], or Chernoff faces [27] being only some well known examples.

In scientific visualization most work on iconic representations has been focused on the visualization of vector and tensor fields. Elementary glyphs such as hedgehog lines or arrows provide a straight-forward, natural, and direct mapping of local vector field properties, like direction and vector magnitude, and have been used for the purpose of visualizing flow fields for at least three centuries [74]. However, the need for visualizing large three-dimensional computer simulations of complex multi-variate fields has given rise to the development of a multitude of application-tailored glyphs. De Leeuw and van Wijk [35] developed a sophisticated glyph they call the flow probe that visualizes additional local attributes, such as the Jacobian, divergence, and other derived properties of a flow field. Even more abstraction was achieved by Globus et al. [60] who use composite arrow glyphs to depict the location and classification of critical points in a vector field in order to visualize the topology of the field. Glyphs have been also used for

generic feature visualization. Silver et al. [195] and Post et al. [165], for example, use ellipsoidal glyphs in feature-based flow visualization.

A detailed discussion on the use of glyphs for vector and tensor field visualization as well as a complete classification scheme for generalized vector and tensor glyphs can be found in the work of Hesselink and Delmarcelle [79, 38]. In the following, however, we will focus on tensor glyphs and in particular on techniques for visualizing second-order tensor fields, i.e. 3^2-variate fields. A number of different glyph representations have been proposed for visualizing such tensor data, in particular stress and strain rate tensor data from engineering mechanics and computational fluid dynamics simulations and, more recently, for measured data originating from medical diffusion tensor imaging (DT-MRI).

6.1 Tensor Glyphs

In the following we will consider three-dimensional real symmetric, second-order tensor fields, i.e. 6-variate variables. In fact, most work on the visualization of tensor fields is focused on symmetric tensors since most practical applications yield symmetric tensors anyway, for example DT-MRI scans and stress and strain-rate fields from FEM or CFD simulations. However, in general every three-dimensional second-order tensor $T = (t_{ij})$, can be represented by the sum

$$T = S + A \tag{6.1}$$

of a symmetric $S = (s_{ij})$ and an antisymmetric tensor $A = (a_{ij})$, where

$$s_{ij} = \frac{1}{2}(t_{ij} + t_{ji}) \quad \text{and} \quad a_{ij} = \frac{1}{2}(t_{ij} - t_{ji}).$$

Which can again be represented by a total of nine independent scalar values—six scalars for the symmetric part and three scalars for the antisymmetric part.

Each symmetric tensor can be further factored in a product

$$S = R\Lambda R^{-1} \tag{6.2}$$

of a diagonal matrix Λ and an orthogonal matrix R [184]. In this so-called *eigen decomposition* the diagonal elements of Λ are the eigenvalues λ_i of S whereas R describes the transformation of the standard basis into the eigenvector basis of S, i.e. the columns of R equal the eigenvectors ζ_i of S.

Further, we will assume that all tensors have full rank, i.e. $|S| \neq 0$ or equally $\lambda_i \neq 0, \quad i = 1, \ldots, 3$. Tensors that do not fulfill this criterion will be called degenerate in the following. Note that this definition differs from the standard definition of tensor degeneracy found in the literature. Normally, a degenerate

tensor is defined as a tensor for which no complete eigenvector basis exists [184]. Since, symmetric tensors always feature a complete eigenvector system this is always true in our case. In the tensor visualization literature, especially in tensor field topology related papers, yet another definition of tensor degeneracy can be found [80]. A second-order tensor in this context is already defined as being degenerate if an eigenvalue has a multiplicity greater than one. Note that this is more general than the previous definition, since this does not necessarily imply a reduced eigenvector system. That is, a degenerate tensor in this sense is a tensor that is isotropic, i.e. every coordinate system is an eigenvector system of the tensor, in the case of a triple eigenvalue, or at least partly isotropic, i.e. every orthogonal coordinate system with one axis defined by the eigenvector corresponding to the single eigenvalue, in case of a double eigenvalue.

The simplest glyph widely used in tensor field visualization is the ellipsoid. The so-called *Lamé stress ellipsoid* has a long tradition in tensor field visualization. Originally devised in the theory of elasticity [133], the stress ellipsoid represents the distortion of an infinitesimal volume element. Its orientation and principal axes encode the directions and magnitudes of the principal stress states defined by the eigenvectors and eigenvalues of the stress tensor. In other words, the eigenvalues and eigenvectors of a tensor are identified with the shape parameters of an ellipsoid, where the absolute values of the eigenvalues correspond to the lengths of the principal axes of the ellipsoid and the eigenvector system specifies its orientation with respect to the standard basis. Thus, a tensor ellipsoid is defined by rotating a sphere scaled by Λ according to the rotation matrix R.

Due to visual ambiguities—depending on the viewing conditions the visual impression of a prolate ellipsoid cannot be distinguished from the image of a spherical or oblate ellipsoid—caused by the smooth surface of the ellipsoid, often other glyph shapes with pronounced feature lines, which are easier to discern, have been used. This includes boxes, octahedra, and cylinders [167, 226]. However, there are other problems associated with these glyphs. Although cylinders are very well suited to depict tensors with a single distinguished eigenvalue, such as $|\lambda_1| \gg |\lambda_2| \simeq |\lambda_3|$, i.e. those tensors represented by elongated tensor ellipsoids, they fail in the case of planar shapes with two or no distinguished eigenvalues, i.e. oblate shapes with $|\lambda_1| \simeq |\lambda_2| \gg |\lambda_3|$ and spherical shapes with $|\lambda_1| \simeq |\lambda_2| \simeq |\lambda_3|$ respectively. Furthermore, there is no continuity in shape changes when, for example, moving continuously from linear to planar shapes. Boxes or cuboids in turn are misleading in case the tensor is isotropic, i.e. $|\lambda_1| \simeq |\lambda_2| \simeq |\lambda_3|$, or has at least two close eigenvalues. In these cases there exists no unique eigenvector system and, therefore, the missing rotational symmetry of the glyphs suggests an anisotropy that is inexistent.

A solution that combines the positive aspect of strong visual cues provided by the pronounced edges of cylindrical and cuboid glyphs with the smooth surface

and the symmetry properties of the ellipsoid are superquadric tensor glyphs [47, 48, 95, 87]. However, they are much more complex to generate in contrast to simple ellipsoids.

Also composite glyphs consisting of a combination of basic shapes have been proposed in order to alleviate the problem of visual ambiguities. Haber [69, 70], for example, uses a combination of an elliptical disk and a cylinder aligned with the major eigenvector for visualizing the stress tensor fields in fracture propagation simulations. Similarly, Westin et al. [222] propose a composite shape made of separate ellipsoids for the linear, planar, and spherical components of the tensor.

All of the aforementioned glyphs share the common property that they can only depict the absolute value of the eigenvalues. Hence, they are most useful for visualizing positive definite tensors, which for example is the case in diffusion tensor MRI data, as it is discussed in Section 6.5.1). Many other practical problems, such as the visualization of stress and strain rate fields, however, require the visualization of tensors having also negative eigenvalues. Using the aforementioned glyphs does in those cases not convey important tensor properties.

Therefore more complex glyphs are required in that cases. One possibility to graphically represent indefinite tensors is the *Cauchy stress quadric* [133] defined by

$$\mathbf{x}^{\mathsf{T}} S \mathbf{x} = \pm k^2, \quad k \neq 0. \tag{6.3}$$

In principal, this glyph is very similar to the tensor ellipsoid. It also defines a second-order algebraic surface based on tensor properties. In particular, if S is positive definite or negative definite, the quadric surface, again, is an ellipsoid. In case one or two eigenvalues are negative the resulting glyph is the union of a one-sheeted and a two-sheeted hyperboloid. That is, in contrast to the stress ellipsoid, the extent of the stress quadric is not finite in general. Therefore, in order to use it as a glyph it has to be truncated artificially. Furthermore, the shape of the stress quadric might be misleading. In case of the tensor ellipsoid the lengths of the glyph's principal axes are proportional to the magnitudes of the principal stress states. In case of the stress quadric, however, the squared distance r^2 from the center $\mathbf{C}$ to a point $\mathbf{P}$ on the quadric surface is inversely proportional to the normal stress

$$N = \mathbf{n}^{\mathsf{T}} S \mathbf{n}, \tag{6.4}$$

acting in the direction $\mathbf{n} = \frac{\mathbf{CP}}{|\mathbf{CP}|}$ [55]. This can be easily seen by substituting $\mathbf{x} = r\mathbf{n}$ in Equation (6.3). Consequently, this is also the case for the principal stress directions ζ_i. Figure 6.1 gives a comparison of the visualization of a symmetric tensors by ellipsoid glyphs and by stress quadric glyphs. In both cases the same

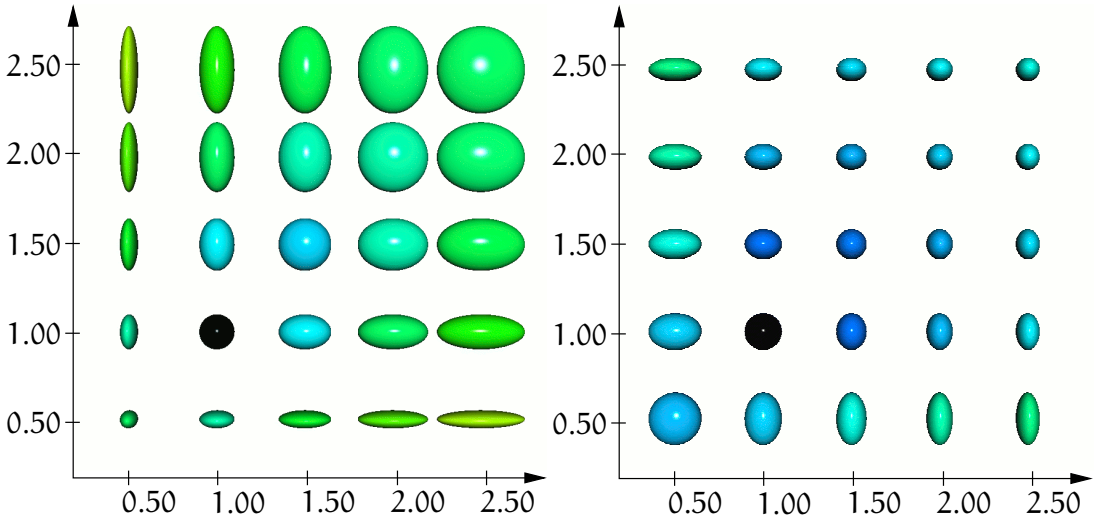


Figure 6.1: Comparison of stress ellipsoid (left) and stress quadric glyph (right) under the same viewing conditions. Glyphs corresponding to normal stress levels varying between 0.5 and 2.5 are shown.

25 tensors

$$S = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \sigma_1, \sigma_2 \in \{0.5, 1.0, 1.5, 2.0, 2.5\}$$

are depicted. However, in the first case the proportions of the ellipsoid, i.e. the lengths of the ellipsoid's principal axes, match the ratio of the principal stress components acting in the directions of its principal axes, while in the second case, although the tensor is represented by the same basic ellipsoidal shape, the lengths of the principal axes are inversely proportional to the square of the principal stress components. This example clearly illustrates the drawback of the stress quadric being not as intuitive as the tensor ellipsoid regarding the representation of the absolute value of the principal stress components, i.e. eigenvalues of the tensor. The properties of the stress quadric will be discussed in depth in Section 6.3.6.

Another possibility that has been proposed by Wünsche [233] is to visualize the signs of the eigenvalues by dividing the ellipsoid into six hemispherical segments and coloring them according to the signs of the eigenvalues.

Beyond these simple glyphs much more complex shapes, like the *Reynolds glyph* [148] and the *HWY glyph* [76], can be found in the literature. Both glyphs are specifically designed to provide improved visualization of specific stress tensor properties. However, their mathematical representations are not as readily amenable to a GPU-based implementation as the previously mentioned quadric

surfaces.

In this chapter a point-based approach for glyph-based visualization of three-dimensional symmetric, second-order tensor fields will be presented. Both the rendering of ellipsoidal glyphs as well as a new quadric glyph shape derived from the stress quadric, which allows the effective visualization of indefinite tensors, will be discussed. This new quadric glyph is more intuitively interpretable than the original stress quadric glyph in the sense that its extents are proportional to the tensor's eigenvalue magnitudes.

The chapter will be concluded by several applications of tensor glyph visualizations. In particular, these are examples of DT-MRI diffusion tensor visualization, the visualization of strain-rate fields derived from fluid dynamics simulations, and the illustrative visualization of vector fields, in particular magnetic fields.

6.2 Basic Approach and Related Work

Traditionally, glyphs are rendered either by raytracing of an implicitly defined shape or by tesselating them into a triangle mesh subsequently rendered using the graphics processor. Although raytracing produces images of very high quality it is in general too slow in order to be suitable for interactive visualization. On the other hand, the number of triangles required for a suitably smooth tesselation of the glyph surface is very high. Therefore, rendering several hundred thousands or even millions of tesselated tensor glyphs is again a performance problem due to the huge amount of geometry data that has to be processed in the vertex and primitive processing stages of the rendering pipeline. Even when OpenGL display lists or vertex array objects are employed to cache the geometry in GPU memory or instanced rendering is used, the frame rates that can be achieved for reasonably smooth tessellated ellipsoids are far from being interactive. Furthermore, since it is often not possible to select or filter the data directly on the GPU, e.g. for time-dependent data sets, it is not possible to store the data in video memory. Instead, it has to be streamed to the GPU for every rendered frame. Hence, the graphics interface bus might be a second bottleneck in addition to vertex processing.

This problems can be illustrated by the following simplified theoretical example. Assume we like to render 250,000 ellipsoidal glyphs, roughly corresponding to a 64^3 regular tensor grid. This can be accomplished by tesselating the unit sphere and scaling and rotating it accordingly. Tesselating a sphere requires $3 \cdot 4^{\mathrm{l}}$ triangles when using a standard uniform recursive subdivision of level l based on a regular tetrahedron. On the other hand, the peak geometry processing performance of a high-end NVIDIA Quadro 5600 GPU is specified with 300 million triangles per second which is likely to be achieved only when rendering pretrans-

formed flat-shaded triangles. Nevertheless, this allows only at most two subdivision levels or 48 triangles per ellipsoid to be rendered at interactive frame rates of at least 20fps regardless whether the data is streamed from main memory, is resident in GPU memory via vertex buffer objects, or geometry instancing techniques are employed. This rather coarse tesselation, however, is not sufficient for high-quality visualizations.

One possibility to alleviate this problem is to use view-dependent level-of-detail (LOD) techniques [134] to adapt the number of polygons, i.e. the tesselation of a glyph, according to the actual distance of the glyph to the viewer. This way smooth surfaces can be guaranteed for glyphs close to the viewer while distant glyphs can be represented by coarsely tesselated geometry. However, such an approach involves a large CPU processing overhead for LOD selection and still has a rather high geometry processing load for a large number of glyphs.

So, the goal of this chapter is to develop a GPU-based glyph rendering technique that achieves high-quality surface visualizations comparable to images rendered off-line using raytracing but without the cost involved in using tesselated geometry. That is, to reduce the geometry processing load and at the same time reduce both storage requirements and bandwidth requirements for uploading or streaming glyph data from main memory to the GPU.

The solution proposed here [100] employs a single point primitive in combination with a set of vertex and fragment shaders to render the visual representation of the glyph shape by means of a GPU-based ray casting of the implicit surface description of the quadric glyph surface on a per-fragment basis. Hence, the presented approach reduces the amount of information that has to be transferred over the graphics bus to the GPU or has to be stored in GPU memory to the necessary minimum. In particular, the required geometric data per rendered glyph object is broken down to the parameters that define the spatial location, the orientation, and the shape of the glyph, in other words the surface center position $\mathbf{c}$ of the glyph, a quaternion $\mathbf{q}$ representing the rotation matrix R that describes the orientation of the local coordinate system spanned by the principal axes of the glyph, and a vector $\mathbf{h} = (\lambda_1, \lambda_2, \lambda_3)^\mathsf{T}$ that represents the shape parameters given by the eigenvalues. These three properties can be easily encoded into three vertex attributes, such as vertex position, normal vector, and vertex color attribute, of an OpenGL point primitive. Thus, no more than ten floating point values, or 40 bytes, per tensor glyph have to be transferred over the system bus to the GPU. In the following sections the GPU-based rendering algorithm which allows to compute the perspectively correct projection of Phong-shaded ellipsoidal and hyperboloidal shapes from this data will be described in detail.

This technique is closely related to the widening field of point-based rendering techniques [171]. The rendering of point-sampled surface representations has become a very popular field of computer graphics research in the last years [125,

180, 159, 243]. As these techniques do not rely on the availability of connectivity information or topology they are perfectly suited for dealing with the massive amount of data acquired from complex, real-world 3D objects by scanning devices, such as laser scanners [124], without the need of reconstructing a surface mesh. Furthermore, since polygonal complexity of geometric models has increased enormously, point-based rendering, possibly in combination with traditional polygon rendering, has also become an interesting alternative to triangle-based rendering [214] as the sub-pixel screen-space footprint of a single triangle does no longer justify the overhead involved in storing mesh connectivity. Naturally, this has led to a number of GPU-based surface-splatting techniques [20, 21, 19] that exploit the programmability of todays GPUs for fast and perspectively correct rasterization of Phong-shaded surface splats. Recently, even a dedicated hardware architecture for point splatting has been presented [225].

On the other hand, the problem of rendering individual glyphs, is inherently different than rendering a point-sampled surface. While the individual surface splats represent small patches of a surface, a 3D glyph is an object in its own right. Therefore, different rendering methods are required.

A first approach for point-based rendering of geometric shapes, in this cases arrow glyphs, was presented by Guthe et al. [65]. They use prerendered views of the glyph stored in a texture map that are appropriately oriented and mapped on view-aligned splats. However, due to the texture-based approach, the glyph shapes are not perspectively correct represented nor is correct depth sorting for overlapping objects guaranteed. The same is true for the precomputed lighting. Furthermore, there are sampling issues regarding the discrete number of orientations that are stored in the texture and due to the limited resolution of the texture maps. A similar approach for rendering ellipsoidal shapes for visualizing molecular data was presented by Max [141].

Other texture-based approaches for rendering spheres [51] or cylinders and helices [5, 75] improve on the depth sorting, at least for orthographic projections, and lighting issues by using a depth map and a normal texture. The achievable rendering quality, though, still suffers from the limited resolution of the texture maps.

The first published technique for rendering perspectively correct ellipsoidal shapes that does not depend on tesselated geometry or pregenerated textures was presented by Gumhold [64]. The splatting approach proposed by Gumhold uses a quadrilateral per ellipsoid, which corner vertices are suitably transformed in the vertex shader so as to enclose the screen-space silhouette of the respective ellipsoid. Exploiting the linear interpolation during rasterization, per fragment attributes encoding the viewer position and the respective view ray are passed to the fragment shader that in turn computes intersections of the view ray with the ellipsoid surface respectively the correct depth and shading information for each pixel

covered by the ellipsoid's screen projection. Thereby, fragments lying outside the silhouette are discarded. This approach, although the basic idea is very similar, differs from the point-based method that will be described in the following, as it uses quadrilateral splats instead of points as the basis for the rendering.

Other similar and related work includes the point-based rendering of molecule data [170, 194] and the splat-based ray casting of streamlines [199] and hyperstreamlines [169].

6.3 GPU-based Rendering

In this section a brief outline of the rendering algorithm will be given. As mentioned before, for each glyph an OpenGL point primitive is rendered, either by specifying the described glyph properties using immediate mode OpenGL calls suitable for interactive streaming of the data or using vertex arrays and vertex buffer objects for reduced CPU and bus transfer overhead. The actual rendering of the particle shapes then takes place using a combination of vertex and fragment shader programs. Starting with the discussion of the glyph surface representation for positive definite tensors that can be represented by ellipsoids, the point-splatting approach will be described, including optimizations like deferred shading and efficient splat geometry computations. The result will then be generalized to the more general stress quadric glyphs in Section 6.3.6.

For both kinds of glyphs, the rough outline of the algorithm is as follows: First, for each glyph the screen area that is covered by the silhouette of its perspective projection has to be determined and the size of the rasterized point primitive adapted accordingly to enclose it completely. This provides the necessary fragments for the following ray-surface–intersection computation, similar to the proxy geometry used for volume rendering in Section 4.2.1. Thereto for each fragment covered by the point primitive a ray from the eye point through that fragment is computed. This eye ray is intersected with an implicit representation of the glyph surface in order to decide whether the fragment belongs to the screen-space projection of the glyph or not. Depending on this decision the fragment is either discarded or the intersection point and the surface normal are computed for shading the resulting output fragment.

6.3.1 Surface Representation

Associated with each ellipsoid there is a local coordinate system (object space), in which its boundary surface can be represented implicitly by the quadratic form

$$\left\{\tilde{\mathbf{x}} \mid \|\tilde{\mathbf{x}}\|^2 = 1\right\}. \tag{6.5}$$

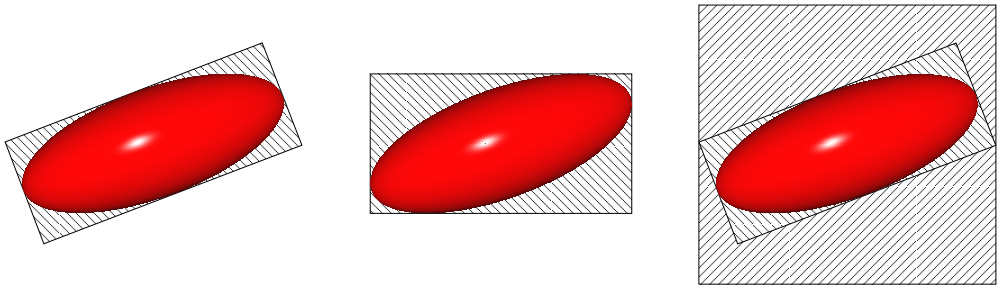


Figure 6.2: The problem of overestimated splat size illustrated. From left to right: silhouette-aligned splat, screen-aligned splat, and point splat.

Note, that all object-space points will be denoted with a tilde in this and the following sections. All other points are meant to be in world space, if not stated explicitly. Starting from Equation (6.5) the world space description

$$\left\{\mathbf{x} \mid \|M^{-1}(\mathbf{x}-\mathbf{c})\|^2 = 1\right\} , \tag{6.6}$$

can be derived, where $M = RH$ is a symmetric, positive definite matrix given by the scaling matrix $H = \mathrm{diag}(\mathbf{h})$ and the rotation matrix $R = R(\mathbf{q})$ that describes the orientation with respect to the local coordinate system of the ellipsoid. $\mathbf{c}$ is the world space position of the glyph's center point. Thus, an affine mapping $\mathbf{x} = M\tilde{\mathbf{x}} + \mathbf{c}$ from the local parameter space of the ellipsoid spanned by the eigensystem of the tensor to the world coordinate system can be defined.

6.3.2 Point Splatting

Since only a single point primitive is drawn, the screen space footprint covered by the perspective projection of the ellipsoid's shell has to be determined and the screen-space size of the rendered base point has to be adapted accordingly. This is realized in a vertex shader program by projecting the eight corners of the object-aligned bounding box into clip space. Since OpenGL points are always square, the final point size is defined by the longest edge of the axis-aligned rectangle enclosing the projection. This is obviously the major drawback of the point-splatting approach. As is illustrated in Figure 6.2 a large number of unnecessary fragments are generated that require costly computation in the subsequent per-fragment ray casting procedure although they have to be finally discarded. This is most disadvantageous in case the ellipsoid projects to a rather large, *non-axis-aligned* screen area or very elongated ellipsoids are drawn. Thus, a lot of per-fragment computations could be saved if a screen-aligned or silhouette-aligned splat could be used. However, both the silhouette-aligned splat as well as the screen-aligned splat depend on the rendering of quadrilateral splats. This would quadruple the data size

and accordingly result in four times the vertex processing load. Even more, the generation of a truly silhouette-aligned screen-space splat would require the computation of the principal axes of the projection ellipse. This, however, requires significant effort [46] and is not feasible in a shader program.

In addition, all values that are needed for the ray-glyph intersection fragment program and remain constant for all fragments covered by the respective ellipsoid are precomputed in the vertex shader program. Passing these as varying vertex parameters, the values are automatically replicated for each fragment of the rasterized point by the linear interpolation of vertex attributes. Precomputing as many values as possible reduces the rasterization cost, i.e. the operation count, in the subsequent fragment shader program.

An alternative approach to square splat geometries that can be implemented completely in the highly programmable shader pipeline offered by the last generation of graphics processors will be described in Section 6.3.4. This approach employs the newly introduced geometry shader stage of the rendering pipeline in order to generate polygonal, silhouette-aligned imposters from the input data points.

6.3.3 Ray-Glyph Intersection

The next step is to find those fragments that actually belong to the area that is covered by the perspective projection of the ellipsoid and, thus, constitute its surface and to compute the shading required to achieve the three-dimensional impression of the glyph. This step is realized by means of an implicit surface ray casting approach in a fragment shader program.

For each fragment covered by the rasterized point primitive a ray $\tilde{\mathbf{x}} = \tilde{\mathbf{e}} + \lambda\tilde{\mathbf{d}}$ from the eye point through the object-space position of the rasterized fragment on the view plane is computed. Since the origin of the eye ray in object space, $\tilde{\mathbf{e}} = M^{-1}(\mathbf{e} - \mathbf{c})$, is constant for all fragments of the projection it has to be computed only once per ellipsoid, which can also be done in the vertex program. In contrast, the ray direction $\tilde{\mathbf{d}}$ has to be computed for each fragment separately. Because only a single point is rendered it is not possible, e.g., to exploit the linear interpolation of vertex attributes during the rasterization for the computation of $\tilde{\mathbf{d}}$, as it is done by Gumhold [64]. Instead the eye ray direction has to be computed from the current viewing parameters by unprojecting the 2D clip-space coordinates of the fragment. This can be easily accomplished by applying the inverse viewport transformation to the fragment's screen-space coordinate followed by a multiplication with the inverse modelview-projection matrix available as a fragment shader state variable and the transformation into the object coordinate system of the ellipsoid.

Then, all fragments of the point splat can be classified to lie either inside or

outside the silhouette of the projection by intersecting the eye ray with the implicit representation of the ellipsoid. Intersecting a straight line with an ellipsoid is not difficult, but it is even simpler when working in the local coordinate system of the ellipsoid. Since all necessary transformations are affine, and therefore preserve straight line segments, it is quite obvious to transform the eye ray into object space and to do a simple ray-sphere intersection computation. Then, the actual intersection is computed straightforwardly. Inserting the ray equation $\tilde{\mathbf{x}} = \tilde{\mathbf{e}} + \lambda\tilde{\mathbf{d}}$ into the object-space expression $\|\tilde{\mathbf{x}}\| = 1$ of the ellipsoid yields the condition

$$\tilde{\mathbf{d}}^T\tilde{\mathbf{d}}\lambda^2 + 2\tilde{\mathbf{e}}^T\tilde{\mathbf{d}}\lambda + \tilde{\mathbf{e}}^T\tilde{\mathbf{e}} - 1 = 0 \,, \tag{6.7}$$

which only then has real solutions if the determinant

$$D = (\tilde{\mathbf{e}}^T\tilde{\mathbf{d}})^2 - \tilde{\mathbf{d}}^T\tilde{\mathbf{d}}(\tilde{\mathbf{e}}^T\tilde{\mathbf{e}} - 1) \tag{6.8}$$

is greater or equal to zero. Depending on the sign of D the fragment is either discarded, i.e. killed in fragment shader terminology, or the intersection point and the ellipsoid normal necessary for shading the resulting pixel are computed. Solving (6.7) for the actual intersection parameter

$$\lambda_s = \frac{-\tilde{\mathbf{e}}^T\tilde{\mathbf{d}} - \sqrt{(\tilde{\mathbf{e}}^T\tilde{\mathbf{d}})^2 - \tilde{\mathbf{d}}^T\tilde{\mathbf{d}}(\tilde{\mathbf{e}}^T\tilde{\mathbf{e}} - 1)}}{\tilde{\mathbf{d}}^T\tilde{\mathbf{d}}} \tag{6.9}$$

allows the computation of the intersection point $\tilde{\mathbf{s}} = \tilde{\mathbf{e}} + \lambda_s\tilde{\mathbf{d}}$. Note that although Equation (6.7) in general will have two distinct solutions we are only interested in the smaller value and as $\tilde{\mathbf{d}}^T\tilde{\mathbf{d}} > 0$ this solution is well-defined.

Since the whole computation takes place in the local parameter space of the ellipsoid the position vector of the intersection point is also identical to the normalized surface normal. With this information per-fragment correct Phong-lighting of the ellipsoid surface can be computed. There are only two things missing: The light vector pointing from the point of intersection to the light source position and its reflection about the surface normal, which are both easy to compute.

Depth Correction As a last step, the correct depth sorting of the rendered objects has to be ensured. For each fragment, the correct depth value has to be computed and written to the z-buffer. Thereto the point of ray-surface intersection $\tilde{\mathbf{s}}$ is transformed to world space and the modelview and projection transforms are applied accordingly. After that, perspective division and depth range mapping yields the corrected depth value for the fragment that guarantees the correct depth sorting of both the ellipsoid glyphs and traditionally rendered geometry.

6.3.4 Polygonal Imposters

The main drawback of the point-based approach described in the previous sections is the creation of many unnecessary fragments due to the fact that OpenGL points are always square.[1] As mentioned earlier, the overhead, especially for very elongated ellipsoids, is very high and can in practice easily exceed 80% of all generated fragments.

In this section a technique will be introduced that aims to avoid the generation of superfluous fragments by choosing a better approximation of the ellipsoid's projected silhouette. Note that the thus projected silhouette is again a quadratic form; in this case an ellipse. Obviously, the rectangle is a bad choice for the bounding polygon of a circle or an ellipse. A much tighter fitting bounding geometry is a regular polygon circumscribing the ellipse. However, as stated before, uploading more vertices is no solution. Therefore, a solution is required that combines the advantages of a tight fitting splat geometry with the inexpensive upload and storage of the point-based approach. Fortunately, the newly introduced geometry shader stage in the rendering pipeline of modern GPUs allows the generation of geometry on-the-fly. Thus, a complex bounding geometry such as a n-sided polygon can be generated using a single point as input primitive.

The basic idea is to use the object space definition of the ellipsoid's silhouette as described by Gumhold [64] in order to generate a polygonal billboard or imposter. Given an ellipsoid in world space its silhouette as it is seen from the eye point $\boldsymbol{e}$ is given by all points on the ellipsoid's surface where the tangent cone with tip $\boldsymbol{e}$ touches the surface and is orthogonal to the surface normal $\mathbf{n}$, i.e.

$$\left\{\mathbf{x} |\ \|M^{-1}(\mathbf{x}-\mathbf{c})\|^2 = 1 \wedge \mathbf{n}^{\mathsf{T}}(\mathbf{x}-\boldsymbol{e}) = 0\right\} \tag{6.10}$$

or equivalently in object space

$$\left\{\tilde{\mathbf{x}} |\ \|\tilde{\mathbf{x}}\|^2 = 1 \wedge \tilde{\mathbf{x}}^{\mathsf{T}}(\tilde{\mathbf{x}}-\tilde{\boldsymbol{e}}) = 0\right\}\ . \tag{6.11}$$

Exploiting the geometrical relationships illustrated on the left of Figure 6.3 and some simple trigonometry Gumhold derived the following parametric description of the silhouette circle in object space:

$$\left\{\tilde{\mathbf{x}} |\ \tilde{\mathbf{x}} = \frac{1}{\tilde{\boldsymbol{e}}^2}\tilde{\boldsymbol{e}} + \sqrt{1-\frac{1}{\tilde{\boldsymbol{e}}^2}}\left(\sin\alpha\tilde{\mathbf{u}} + \cos\alpha\tilde{\mathbf{v}}\right)\right\}, \tag{6.12}$$

[1] This is not true in general. Antialiased OpenGL points are rendered as filled circles. However, the circle is also not a perfect bounding geometry for an ellipse and, furthermore, rendering antialiased points is orders of magnitudes slower than rendering plain points and has strict restrictions regarding the allowed point sizes.

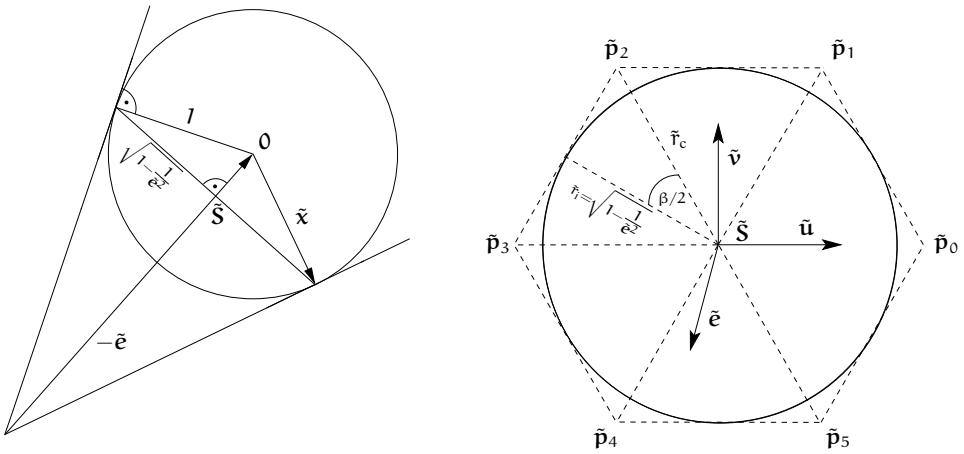


Figure 6.3: Rendering ellipsoids using an n-sided regular polygon as splat geometry. The left figure illustrates the object-space silhouette-point computation. On the right the computation of the vertex positions of the actual splat geometry is sketched.

where $\tilde{\mathbf{u}}$ and $\tilde{\mathbf{v}}$ span an orthonormal coordinate system in the plane E perpendicular to the eye vector $\tilde{\mathbf{e}}$. One possibility to compute this local coordinate system is, e.g., by defining a vector

$$\hat{\tilde{\mathbf{e}}} = \begin{cases} \begin{pmatrix} \tilde{e}_2 \\ \tilde{e}_0 \\ -\tilde{e}_1 \end{pmatrix}, & \text{if} \quad \tilde{e}_0 \neq \tilde{e}_1 \\ \\ \begin{pmatrix} \tilde{e}_2 \\ \tilde{e}_0 \\ 0 \end{pmatrix}, & \text{if} \quad \tilde{e}_0 = \tilde{e}_1 \end{cases}, \tag{6.13}$$

which is guaranteed to be linearly independent of $\tilde{\mathbf{e}} = (\tilde{e}_0, \tilde{e}_1, \tilde{e}_2)^T$. The sought-after right-handed system is then given by

$$\tilde{\mathbf{e}}, \quad \tilde{\mathbf{u}} = \tilde{\mathbf{e}} \times \hat{\tilde{\mathbf{e}}}, \quad \text{and} \quad \tilde{\mathbf{v}} = \tilde{\mathbf{e}} \times \tilde{\mathbf{u}}. \tag{6.14}$$

Given the silhouette circle's center $\tilde{\mathbf{s}} = \frac{1}{\tilde{e}^2}\tilde{\mathbf{e}}$ and its radius $\tilde{r}_i = \sqrt{1 - 1/\tilde{e}^2}$ the vertex positions defining a regular n-sided circumscribing splat polygon can be easily computed. This is also illustrated in Figure 6.3. The circumradius $\tilde{r}_c$ of the polygon can be directly derived from its inradius $\tilde{r}_i$:

$$\tilde{r}_c = \frac{\tilde{r}_i}{\cos\frac{\beta}{2}} = \frac{\sqrt{1 - \frac{1}{\tilde{e}^2}}}{\cos\frac{\pi}{n}}. \tag{6.15}$$

This yields the n vertices $\tilde{\mathbf{p}}_0, \ldots \tilde{\mathbf{p}}_{n-1}$ of the splat polygon, which are given by

$$\tilde{\mathbf{p}}_k = \tilde{\mathbf{s}} + \tilde{r}_c \left(\sin \frac{k\pi}{n} \tilde{\mathbf{u}} + \cos \frac{k\pi}{n} \tilde{\mathbf{v}} \right), \quad k = 0 \ldots n-1. \tag{6.16}$$

Transforming the vertices to world space and applying the modelview-projection transformation finally completes the splatting of the imposter geometry. The implementation is then straightforward. The geometry shader program takes a single point primitive attributed with glyph center, orientation, and lengths of the principal axes as its input, computes the clip-space coordinates of the circumscribing splat polygon's vertices according to Equation (6.16), and emits a triangle strip of $n-2$ triangles defined by those vertices. For efficiency reasons hard-coded shader programs for different tesselation levels have been used.

What remains is the question regarding the quality of the approximation or, in other words, how large is the actual reduction in redundantly created fragments when using a polygonal imposter instead of a point. First some theoretical considerations. Given the silhouette circle in object space, the regular n-sided polygon provides the best possible approximation. Furthermore, since affine transformations preserve area ratios this is equally valid for the view-space silhouette after scaling, rotation, and view transformation has been applied. That is, the resulting ellipse will be still the inellipse[II] of the imposter polygon. The perspective projection, however, is problematic. Although there are still n osculation points of the projected ellipse and the polygon, i.e. the ellipse is still inscribed to the polygon touching all its edges, in general the silhouette is no longer an inellipse of the polygon. In other words the projected area of the splat will be larger than necessary. As mentioned in Section 6.3.2, finding the perfect screen-space polygon requires rather complex and costly computations and is thus not feasible. On the other hand, in practice extreme perspectives are only seldom found in visualization, e.g. the field-of-view angle is only rarely chosen much wider than 60 to 70 degrees and also the distance to the viewer is typically not too small. Thus, the perspective distortion caused by the screen-space projection of the polygon and the thereby created additional fragments are negligible in most practically relevant cases.

In order to investigate the actual overhead of redundantly generated fragments a simple experiment has been conducted. Assuming the perfect screen-space circumscribing polygon would be computed for a certain ellipse, the percentage of overhead caused by using a suitably affine transformed n-sided regular polygon would be the same as for a unit circle and the related n-sided regular polygon.

[II] The inellipse of a convex polygon is defined as the ellipse of maximum area inscribed to the polygon.

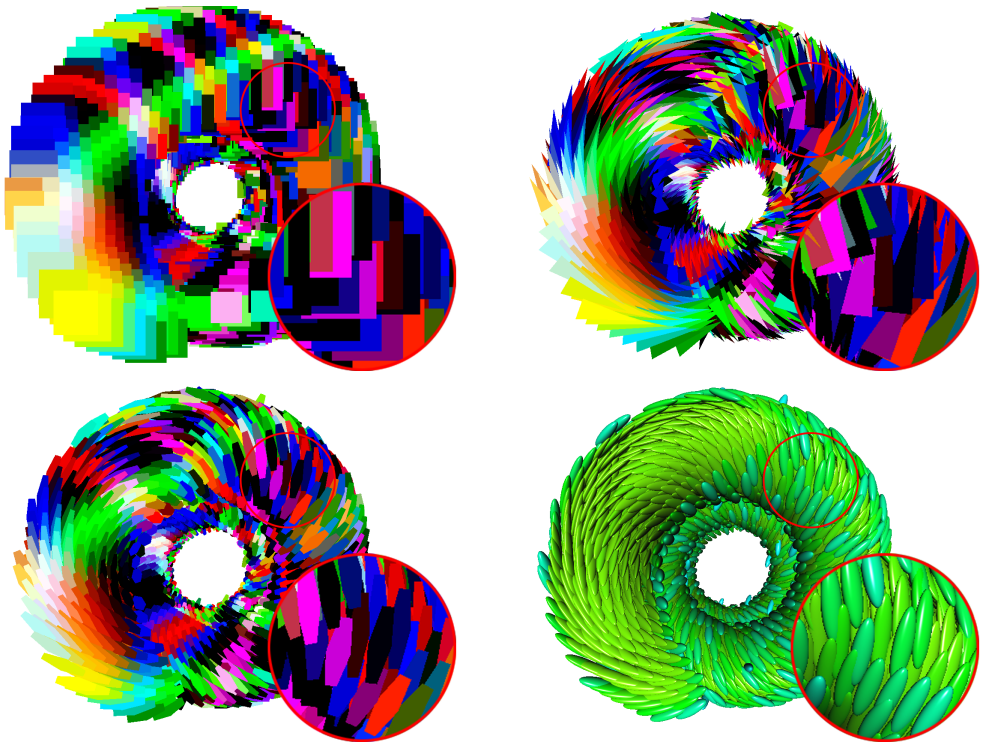


Figure 6.4: Comparison of different splat geometries. Top, left: Point splats. Top, right: 6-sided polygons. Bottom, left: Triangles. Bottom, right: Final rendering result.

Which can be easily computed as

$$1 - \frac{A_{\text{circle}}}{A_{\text{poly}}} = 1 - \frac{\pi}{\frac{1}{2}n\left(\frac{1}{\cos\frac{\pi}{n}}\right)^2 \sin\frac{2\pi}{n}} = 1 - \frac{\pi}{n\tan\frac{\pi}{n}}. \tag{6.17}$$

For the simple test data set of a twisting helix[III] shown in Figure 6.4 the amount of superfluous fragments generated using up to six-sided polygonal imposters matched very good with the theoretically achievable values as can be seen in Table 6.1. The first row shows the total number of generated fragments when using the respective splat geometries. In the second row the number of redundantly generated fragments is shown. The third and fourth row quantify the theoretical overhead according to Equation (6.17) and the actually measured overhead according to the first two rows of the table. Last the performance that has been achieved

[III]Synthetic tensor data set adapted from a data set kindly provided for download by Gordon Kindlmann (http://www.sci.utah.edu/~gk/DTI-data/).

Table 6.1: Statistic of redundant fragments and rendering performance when using different splat geometries for the data set shown in Figure 6.4.

	points	3-sided	4-sided	5-sided	6-sided
fragments total	43,688,332	11,310,792	8,703,180	7,902,080	7,535,767
fragments discarded	36,849,965	4,472,614	1,865,080	1,064,066	697,875
theoretical overhead	-	39.540%	21.460%	13.519%	9.310%
measured overhead	84.35%	39.543%	21.43%	13.466%	9.261%
performance	50.2fps	106.9fps	106.7fps	104.3fps	110.5fps

using an NVIDIA GeForce 8800 Ultra based graphics board on a 1200×860 pixels viewport is shown. In this example only about 12,000 of the most elongated ellipsoids found in this data set are visualized. What may be surprising is the fact that in most cases the measured overhead is even smaller than what would be theoretically possible. This phenomenon must be attributed to discretization artifacts due to the discrete nature of the framebuffer.

Since the geometry shader stage, at least for the first generation of graphics hardware that features this functional unit, is very output sensitive, i.e. the more vertices are emitted and the more attributes are associated with each emitted vertex the lower is the throughput that can be achieved [211], in practice there is no sense in using a very fine tesselation of the silhouette. The overhead caused by creating this additional geometry and processing it in the subsequent primitive processing stages can pretty well outweigh the performance gain anticipated in the fragment processing stage. Actually, this can be seen in Table 6.1. Using triangles as splat geometry is more efficient than using a quadrilateral or pentagon although for the pentagon only one third of redundant fragments are created compared to the triangle. Therefore, a trade-off has to be made between the creation of superfluous fragments and the cost of extra primitive processing. As can be seen in Table 6.1 already the triangle provides a significant reduction in redundantly generated fragments and accordingly a noticeable increase in rendering performance can be achieved. Furthermore, in practice, using a tesselation with more than six vertices did not provide any noticeable benefit. Even more, the performance gain achieved by using a finer tesselation of the polygon than a triangle depends strongly on the input data. Only for data sets containing a very large number of very thin, elongate ellipsoids increasing the number of vertices beyond three or four will provide a measurable performance gain.

Unfortunately, the approach described in this section only works for ellipsoids and does not easily generalize to the more general quadric shapes described in Section 6.3.6.

6.3.5 Deferred Shading

Another optimization that reduces per-fragment computation overhead is deferring the shading computation after depth sorting has been completed. Since for large numbers of objects the depth complexity is often very high, i.e. there is much occlusion taking place, it is beneficial to defer the actual shading of the glyphs as long as possible. Evaluating the shading only once per pixel instead of unnecessarily shading fragments that are subsequently replaced by others can save a huge amount of costly per-fragment computations. Therefore, deferred shading is a common optimization in case of fragment processing bound problems.

Instead of computing the shading for each fragment directly in the aforementioned fragment shader program, the world space position and normal of the intersection point can be stored into two floating point RGBA off-screen render buffers. Then, in a subsequent shading pass, the actual color values for the pixels are computed. This can be achieved by rendering a screen-sized polygon using a fragment shader program that fetches the intersection parameters from textures bound to the respective render buffers holding the results of the previous render pass. Of course, the shading computation is only meaningful for pixels that are actually covered by a glyph. This can, however, be easily guaranteed by applying an appropriate alpha mask channel, for example.

6.3.6 Quadric Glyphs for Indefinite Tensors

Ellipsoidal glyphs, as described in the previous sections, are well suited for visualizing positive definite tensor data. Although this is sufficient for visualizing a large class of important tensors fields, including for example diffusion tensors, elasticity tensors, permeability tensors, or conductivity tensors, this approach is not suitable for visualizing tensors of the general indefinite case, such as stress and strain tensors. Since the ellipsoid can at best be used to visualize the absolute values of the eigenvalues a more appropriate glyph has to be found. As has been detailed in Section 6.1 the stress quadric can be used to depict both direction as well as magnitude and sign of the principal stress components. However, the stress quadric as defined by Equation (6.3) has some drawbacks that complicate its use as a visualization glyph.

Eigendecomposition of S transforms Equation (6.3) in the form

$$\mathbf{x}^T R A D A R^T \mathbf{x} = \pm k^2, \quad k \neq 0 \tag{6.18}$$

where

$$R = (\zeta_1, \zeta_2, \zeta_3)\,,$$
$$A = \mathrm{diag}\left(\sqrt{|\lambda_1|}, \sqrt{|\lambda_2|}, \sqrt{|\lambda_3|}\right), \quad \text{and}$$
$$D = \mathrm{diag}\left(\mathrm{sgn}(\lambda_1), \mathrm{sgn}(\lambda_2), \mathrm{sgn}(\lambda_3)\right).$$

Thus, similar to the discussion in Section 6.3.1, we can define a local object space coordinate system where the surface pair is represented by

$$\left\{\tilde{\mathbf{x}} |\ \tilde{\mathbf{x}}^T D \tilde{\mathbf{x}} = \pm k^2\right\}, \quad k \neq 0. \tag{6.19}$$

Now the quadric shapes can be classified according to the signs of the eigenvalues λ_i of S or the nonzero elements of D respectively. The surface is actually composed of a pair of surfaces differentiated by the sign of the right hand side of the equation. As S is assumed to be nondegenerate with respect to the definition given in Section 6.1, i.e. $\lambda_i \neq 0$ and furthermore $k \neq 0$ there are only two possible combinations. In case S is positive or negative definite, i.e. all eigenvalues have the same sign, these are an ellipsoid and the empty set respectively. Whereas, in case S is indefinite, the quadric equation describes a pair made of a one-sheeted hyperboloid and a two-sheeted hyperboloid. This illuminates the major problems of the stress quadric approach. First, the quadric surface defined by Equation (6.3) has in general no finite extent. Although positive definite as well as negative definite tensors, again, produce finite ellipsoidal shapes this is not the case for the hyperboloids generated by indefinite tensors. Second, the shapes produced for indefinite tensors are ambiguous with respect to the signs of the eigenvalues; that is, negating all eigenvalues of S or in other words negating S will in fact produce the same pair of quadratic surfaces. Third, since the distance of a point $\mathbf{x}$ on the quadric surface is inversely proportional to the square of the normal stress acting in this direction there is no simple geometrical equivalence between glyph shape and the eigenvalues of the tensor. Thus, the stress quadric surface is not as intuitively interpretable as the ellipsoid. And, last, the surface defined by Equation (6.3) is not closed; the sheets are only asymptotic to each other.

These shortcomings of the stress quadric, however, can be remedied by combining its positive aspect of providing an additional shape for indefinite tensors that can be easily distinguished from the ellipsoid depicting the positive and negative definite cases with the advantages of the ellipsoidal glyph described in the previous sections. In particular these are a finite extent, the proportionality between eigenvalue magnitudes and glyph dimensions, and an implicit description that makes it amenable to a point-based rendering approach. Thus, the goal is to define a glyph that is based on the shape of the stress quadric but is as intuitively interpretable as the ellipsoid.

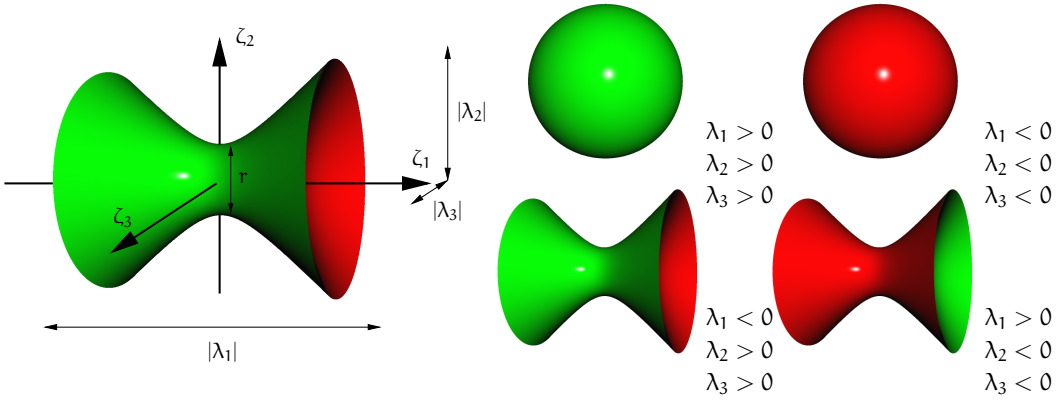


Figure 6.5: The quadric tensor glyph.

By replacing the product ADA in Equation (6.18) by the similar product $H^{-1}DH^{-1}$, where $H = (h_{ii}) = (|\lambda_i|)$ is a scaling matrix similar to the one defined in Section 6.3.1, the proportionality between the glyph's spatial extent and the magnitudes of the eigenvalues. In fact, for a definite tensor this leads directly to the definition of the tensor ellipsoid. Thus, in this case choosing $k^2 = \pm 1$ yields Equation (6.6).

However, the case of an indefinite tensor is more involved. Besides the aforementioned rescaling it is also necessary to clamp the surface to a finite extent and to guarantee a closed surface. Without loss of generality we can assume that the quadric shape in parameter space is given by

$$-\tilde{x}^2 + \tilde{y}^2 + \tilde{z}^2 = \pm k^2. \tag{6.20}$$

All remaining cases can be reduced to this case by symmetry; thus, it is sufficient to investigate this example. Furthermore, since the resulting shape shall be a one-sheeted hyperboloid the right-hand side of Equation (6.20) must be positive. In other words the basic shape of the glyph is defined by the quadratic form

$$Q : -\tilde{x}^2 + \tilde{y}^2 + \tilde{z}^2 = k^2. \tag{6.21}$$

Next, we have to ensure the correct scaling behavior of the glyphs with respect to the eigenvalues magnitudes, see Figure 6.5 (left). This requires k^2 to be chosen appropriately. In order to achieve this, we start by fixing the radius r of the glyph at its smallest diameter. In other words we demand that

$$Q|_{\tilde{x}=0} : \tilde{y}^2 + \tilde{z}^2 = r^2 \tag{6.22}$$

and accordingly $k^2 = r^2$. On the other hand, the proportionality constraint regarding the eigenvalue magnitudes requires that the maximum extent of the glyph in

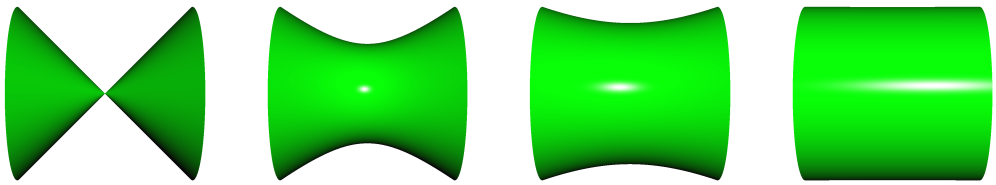


Figure 6.6: The effect of varying the radius r of the quadric tensor glyph. From left to right $r = 0, 1/3, 2/3, 1$.

its local parameter space equals one. Thus, it has to be ensured that the surface is correctly clipped. Using the same argument as before leads to the following equation of the clip plane intersection circle:

$$\tilde{y}^2 + \tilde{z}^2 = 1^2. \tag{6.23}$$

By comparison of coefficients follows $r^2 + \tilde{x}^2 = 1$ and, thus, $\tilde{x} = \pm\sqrt{1 - r^2}$. In other words, points of the hyperboloidal shape have to be culled if $|\tilde{x}| > \sqrt{1 - r^2}$. However, this requires additional scaling by $1/\sqrt{1 - r^2}$ in x-direction in order to ensure the correct extent for all major axes of the glyph. The resulting shapes for both the definite as well as the indefinite case are shown in Figure 6.5 (right).

Note that r provides an extra degree of freedom that allows to map an additional property. This may be either an additional scalar field or a derived tensor property, for instance one of the anisotropy measures discussed in Section 6.4.1. Figure 6.6 shows the effect of varying values of r on the glyph shape. The surface shapes that can be achieved by changing r range from an elliptical double cone for $r \to 0$ to an elliptical cylinder for $r \to 1$.

The thus defined glyph is not a closed surface. Therefore, we have to take into account also the second part of the stress quadric, i.e. the two-sheeted hyperboloid. However, there are two problems. First, the stress quadric per se is not a closed surface since its two subparts are only asymptotic. Second, it is not possible to close the surface just by appropriately choosing the right hand side of Equation (6.20). Thus, we start with the general description of a two-sheeted hyperboloidal quadric

$$\bar{Q} : -a\tilde{x}^2 + b\tilde{y}^2 + c\tilde{z}^2 = -k^2, \tag{6.24}$$

where $a, b, c > 0$ are scaling factors that are yet to be determined. From the fact that the sought-after surface $\bar{Q}$ intersects Q in the clip plane intersection circle defined by Equation (6.23) follows that $c = b$ and

$$\bar{Q}|_{x=\pm\sqrt{1-r^2}} : \tilde{y}^2 + \tilde{z}^2 = 1. \tag{6.25}$$

And, thus,

$$-a(1-r^2)\tilde{x}^2 + b\tilde{y}^2 + b\tilde{z}^2 = -k^2 \tag{6.26}$$

or by comparison of coefficients

$$\frac{-k^2 + a(1-r^2)}{b} = 1. \tag{6.27}$$

Solving for k^2 yields

$$k^2 = a(1-r^2) - b \tag{6.28}$$

for the right-hand–side constant of $\bar{Q}$. Then, from the fact that $\bar{Q}$ shall be a two-sheeted hyperboloid, i.e. $k^2 > 0$, follows

$$a(1-r^2) - b > 0 \tag{6.29}$$

and, thus,

$$a > \frac{b}{1-r^2}. \tag{6.30}$$

Hence, for $a = \frac{b+\epsilon}{1-r^2},\ \epsilon > 0$, we get

$$k^2 = \frac{b+\epsilon}{1-r^2}(1-r^2) - b = \epsilon. \tag{6.31}$$

which yields

$$\bar{Q} : -\frac{b+\epsilon}{1-r^2}\tilde{x}^2 + b\tilde{y}^2 + b\tilde{z}^2 = -\epsilon \tag{6.32}$$

as the equation for the two-sheeted hyperboloid capping the glyph. Which of course has to be also clipped at $\tilde{x} = \pm\sqrt{1-r^2}$.

Theoretically b and ϵ provide further possibility that may be used for varying the glyph shape. For example when varying ϵ the surface will result in a double cone for $\epsilon \to 0$ while for $\epsilon \to \infty$ the result will be a pair of planar surfaces. However, due to the fact that b and ϵ control the shape of the *inner* sections of the glyph and does not affect its silhouette but can be only perceived by rather subtle changes in shading, the effect of varying ϵ and b is not as visually apparent as the effect of varying r. Therefore, it seems appropriate to choose constant values. For simplicity $b = 1$ seems to be a good choice and values of $\epsilon = 0.05 \ldots 0.2$ have been observed to produce satisfactory results.

Thus, taking into account that the ellipsoidal glyph shape should be retained for definite tensors, the object space representation of the general quadric tensor glyph is given by the point set

$$\left\{\tilde{\mathbf{x}} \middle| \left(\tilde{\mathbf{x}}^T D_Q \tilde{\mathbf{x}} = \eta \vee \tilde{\mathbf{x}}^T D_{\bar{Q}} \tilde{\mathbf{x}} = \epsilon\right) \wedge |\tilde{\mathbf{x}} \cdot \mathbf{g}_Q| < \sqrt{1 - r^2}\right\}, \tag{6.33}$$

where

$$\begin{aligned}
\tilde{\mathbf{x}} &= H_Q^{-1} R^T (\mathbf{x} - \mathbf{c}),\\
\eta &= \begin{cases} 1, & \text{if } S \text{ definite} \\ r^2, & \text{if } S \text{ indefinite} \end{cases},\\
D_Q &= \operatorname{sgn}(\operatorname{tr}(D)) D,\\
H_Q &= (h_{ii}), \quad h_{ii} = \begin{cases} \frac{|\lambda_i|}{\sqrt{1-r^2}}, & \text{if } \operatorname{sgn}(\operatorname{tr}(D_Q)) < 0 \\ |\lambda_i|, & \text{if } \operatorname{sgn}(\operatorname{tr}(D_Q)) > 0 \end{cases},\\
\mathbf{g}_Q &= (g_i), \quad g_i = \begin{cases} 1, & \text{if } d_{ii} < 0 \\ 0, & \text{if } d_{ii} > 0 \end{cases},
\end{aligned}$$

and

$$D_{\bar{Q}} = (\bar{d}_{ii}), \quad \bar{d}_{ii} = \begin{cases} -\frac{b+\epsilon}{1-r^2}, & \text{if } d_{ii} < 0 \\ 1, & \text{if } d_{ii} > 0 \end{cases}$$

ensure the correct scaling and clamping of the surfaces. The actual ray casting of the surfaces then is similar to the ellipsoidal case. However, since two surfaces have to be intersected and possibly clipped and the correct intersection point has to be sorted out the computational complexity is much higher than for the simpler purely ellipsoidal case. Besides computing two times a ray-quadric intersection, finding the front-most intersection point that is not clipped is the most expensive part in terms of shader instructions. Instead of 51 `ARB_fragment_program` assembly instructions for the ellipsoid shader, the quadric shader compiles to 241 assembly instructions.[IV] This has a significant impact on the overall rendering performance. For example, when rendering the same artificial data set as has been shown in Figure 6.4 using the quadrics shader the performance drops from approximately 50fps, as has been reported in Table 6.1, to approximately 19fps.[V]

Since the shape of the thus defined glyph is the same with respect to S and $-S$, additional surface properties like color or texture must be used to distinguish these cases. Coloring the glyph surface based on normal stress acting in the direction

[IV] NVIDIA standalone shader compiler *cgc* version 2.0.0.12, *fp40* profile.

[V] NVIDIA GeForce 8800 Ultra based graphics board, 1200×860 viewport, same view as in Figure 6.4

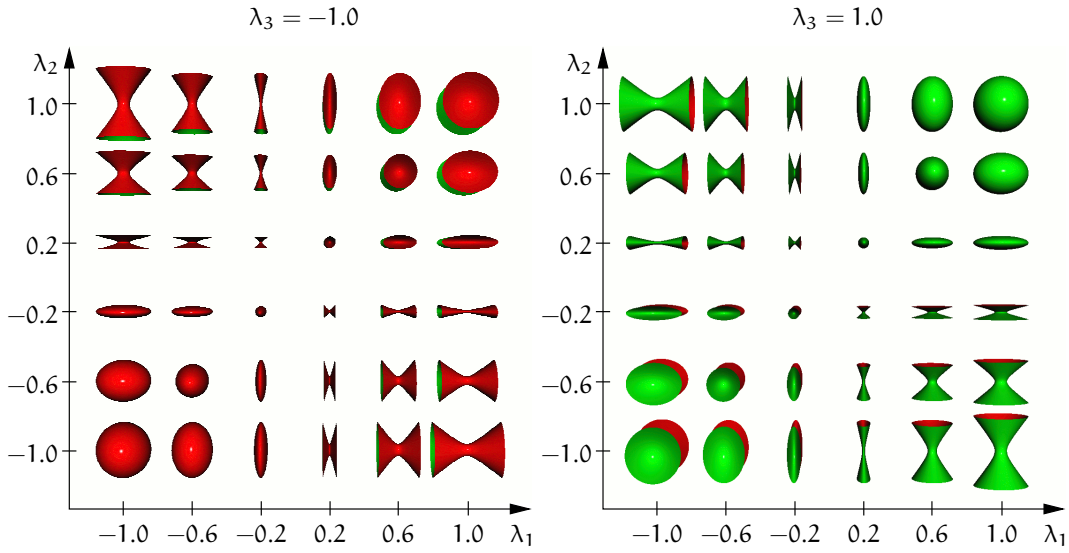


Figure 6.7: Exemplary rendering of quadric tensor glyphs corresponding to different eigenvalue combinations. The glyph radius has been fixed to $r = 0.05$.

of each surface point provides a solution to this problem. Alternatively it is also possible to use a simple color scheme like red for negative and green for positive eigenvalues which borrows its analogy from the coloring typically applied to the poles of permanent magnets and has been used in Figures 6.5, 6.6, and 6.7. Figure 6.7 shows the varying glyph shapes and colorings corresponding to different eigenvalue combinations.

The quadric glyph based on the stress quadric as described above has advantages as well as some disadvantages. First, it provides the means to visualize definite as well as indefinite tensors within a common framework where indefinite tensors can be easily distinguished from definite ones. Compared to more complex glyphs like the Reynolds glyph, the HWY glyph, or the superquadric glyphs mentioned before, however, it can be described implicitly and therefore is easily rendered using a point-based ray casting approach. It clearly depicts the directions and magnitudes of the principal stress and, in contrast to the original stress quadric definition of Equation (6.3), the glyph described in this section is intuitive in the sense that its extents are proportional to the principal stress magnitudes, i.e. to the eigenvalue magnitudes of S.

There are also disadvantages: Like the ellipsoid it does not depict normal stress, which can be achieved by using the Reynolds glyph. Furthermore, as for cylinder glyphs, there are discontinuities in the shape space of the tensor. This can be easily seen in Figure 6.7. However, the discontinuities in this case represent sign changes of the eigenvalues; in other words, fundamental changes in

the nature of the underlying tensor field. Another problem is the use of color for distinguishing the signs of the eigenvalues. This is not a perfect solution with respect to visual clutter when rendering a large number of glyphs. Last, since its rendering requires the intersection computation with a pair of quadratic surfaces in order to determine the correct intersection it is computationally more expensive than the simple ellipsoidal glyphs discussed previously.

Nevertheless, the proposed quadric glyph still allows the interactive visualization of large numbers of tensor samples of an indefinite symmetric second-order tensor field.

6.4 Multi-Field Visualization

Purely glyph-based visualizations suffer from a number of limitations. First, they do not convey the continuity of the underlying tensor field since the glyphs depict only a discretely sampled representation of the continuous field function. Using a very large number of glyphs may alleviate this problem, however, at the cost of visual clutter and high rendering overhead. On the other hand using only few glyphs to highlight the field's behavior in a user-defined region of interest or selected based on specific attributes of the tensor field or another complementary data field often fails in conveying the spatial relationship in the context of the data volume. Opaque and transparent isosurfaces as well as semi-transparent direct volume visualizations, in contrast, allows a qualitative, global visualization at a medium to low abstraction level.

The combination of the sparse geometric glyph visualization technique described in the previous sections with the global volume ray casting approach described in Chapter 4 can, thus, provide both enhanced spatial and semantic context as well as facilitate deeper understanding of the field. Possible additional fields are either derived of the tensor field itself, as will be described in Section 6.4.1, or supplementary scalar fields associated with the tensor field in a multi-field data set.

Since the correct depth information is available in the z-buffer after rendering the glyphs the combination with traditionally rendered geometry or a slice-based volume rendering technique is straightforward. Rendering textured slices after the glyphs have been rendered will take care of the correct depth sorting of glyphs and volume. However, due to the finite distance of the slice planes clipping artifacts will unavoidably degrade the rendering quality.

Fortunately, in a ray casting approach this artifacts can be avoided if the volume integration for each fragment is started or terminated on the actual surface of the front-most glyph. This can be easily achieved by extending the deferred shading approach described in Section 6.3.5. As in the previous multi-pass ap-

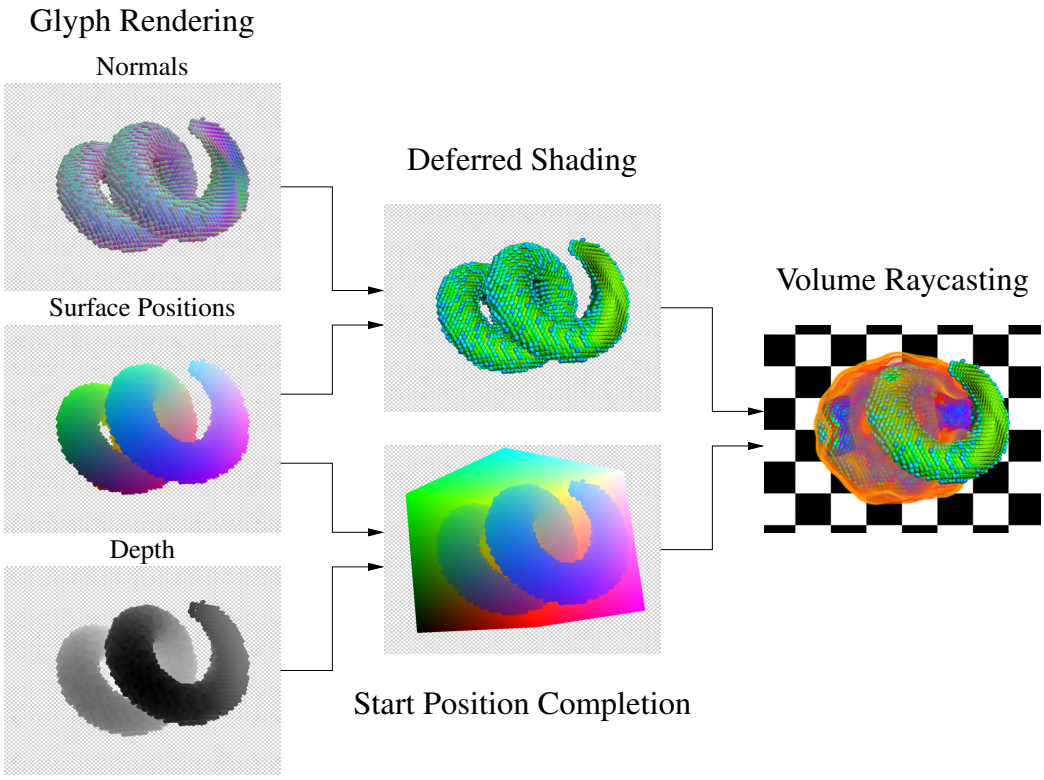


Figure 6.8: Combining glyph-based tensor visualization with volume ray casting.

proach the glyphs are rendered first in a set of off-screen render buffers that can be mapped to textures for subsequent access by a shader. Again, intersection position $\mathbf{s}$ and surface normal $\mathbf{n}$ are stored for all fragments covered by a glyph. Then the shading of the glyphs can be applied and the resulting image stored to an additional texture bound to a render target. In a third pass the actual volume ray casting can be done. For each fragment covered by the volume—indicated by an appropriate alpha mask—a ray is started on the glyph surface position retrieved from the buffer written in the first pass. In contrast to the front-to-back ray casting described in Chapter 4 the volume traversal and color accumulation has to be conducted in back-to-front manner.

However, this will cover only those pixels that are actually covered by the projection of a glyph. Thus, to be precise, an additional intermediate pass is necessary to fill those parts of the start position texture with meaningful values that are covered by the volume's bounding box but not by a glyph. This can be achieved in a manner similar to the hole filling necessary for the empty-space leaping technique discussed in Section 4.5.4. The backfaces of the volume's bounding box are rendered and tested against the depth values of the glyphs. This ensures both

a correct start position for ray traversal if there is no glyph intersecting the ray in question and if the front-most glyph intersected by the ray lies behind the volume. The major steps of the algorithm are shown in Figure 6.8. Special care has to be taken of glyphs lying in front of the volume. In this case the color values of the shaded glyph surface retrieved from the image generated in the second pass have to be returned.

6.4.1 Derived Tensor Quantities

Due to the multi-variate nature of the tensor field, visualizing properties of the tensor field by means of the volume rendering approach described in the previous section requires appropriate contraction of the multi-variate tensor variable. Various derived tensor quantities can be computed that in contrast to the matrix representation of the tensor do not depend on the chosen reference coordinate system and provide quantitative scalar measures of important tensor properties.

Such a reference free quantitative characterization of a tensor can be obtained by studying its invariants. For every tensor $T = (t_{ij})$ these are its rank and the following three scalar invariants

$$
\begin{aligned}
I_1 &= \mathrm{tr}(T) = t_{11} + t_{22} + t_{33} = \lambda_1 + \lambda_2 + \lambda_3, \\
I_2 &= \begin{vmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{vmatrix} + \begin{vmatrix} t_{11} & t_{13} \\ t_{31} & t_{33} \end{vmatrix} + \begin{vmatrix} t_{22} & t_{23} \\ t_{32} & t_{33} \end{vmatrix} = \lambda_1\lambda_2 + \lambda_2\lambda_3 + \lambda_3\lambda_1, \\
I_3 &= |T| = \lambda_1\lambda_2\lambda_3,
\end{aligned}
$$

which define the tensor's characteristic polygon and, thus, can be also expressed in terms of its eigenvalues or eigenvectors. These quantities can provide additional semantic information that can be used in the selection and filtering step or can be visualized in combination with the tensor glyphs—either using the multi-field approach presented in Section 6.4 or as an additional color mapping or texturing on the glyphs' surface in order to emphasize intrinsic properties of the tensor field.

In addition to these general tensor similarity measures more specific application oriented measures have been proposed. In case of diffusion tensors a number of anisotropy indices can be found in the literature that provide a quantitative measure of tensor anisotropy. The important difference between a general symmetric, second-order tensor and a diffusion tensor is the fact that the latter is always positive definite.

Ordering the three positive eigenvalues of a diffusion tensor $D = (d_{ij})$ by decreasing magnitude, i.e. $\lambda_1 \geq \lambda_2 \geq \lambda_3$, several anisotropy ratios can be defined. The simplest and most intuitive being the *ratios of principal diffusivities* [6]

$$
a_{p1} = \frac{\lambda_1}{\lambda_3} \quad \text{and} \quad a_{p2} = \frac{2\lambda_1}{\lambda_2 + \lambda_3}
$$

that describe the ratio between maximum and minor diffusivities.

A set of geometrically motivated quantitative shape measures have been proposed by Westin et al. [223]

$$c_l = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3}$$
$$c_p = \frac{2(\lambda_2 - \lambda_3)}{\lambda_1 + \lambda_2 + \lambda_3}$$
$$c_s = \frac{3\lambda_3}{\lambda_1 + \lambda_2 + \lambda_3}$$

which measure the degree of similarity of the tensor ellipsoid to a line, plane, or sphere respectively. The three quantities add up to unity, i.e. $c_l + c_p + c_s = 1$, and, therefore, define a barycentric space of tensor shapes, with purely linear, purely planar, and purely spherical shapes as extremal cases. This *shape space* can provide the basis, for example, to intuitively define transfer functions for color-mapping tensor glyphs. They have been also employed in direct volume rendering of tensor data [96].

Based on these geometric measures another anisotropy measure, the so-called *anisotropy index* $c_a = 1 - c_s$ can be defined that describes the amount of deviation from the spherical or isotropic case. Alternatively, instead of normalizing by the trace of the tensor also the tensor's Frobenius-norm $\|D\|_F = \sqrt{\mathrm{tr}(D^*D)} = \sqrt{\mathrm{tr}(D^2)} = \sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}$ has been used for normalization [222]. In this case, however, the barycentric characteristic of these measures is lost. Note that all of these measures are susceptible to noise since they are biased by the sorting order of the eigenvalues.

In order to circumvent this issue other anisotropy measures like the *volume ratio index* [161]

$$r_v = \frac{27\,|D|}{\mathrm{tr}(D)^3}$$

and the more complex *relative anisotropy*

$$a_r = \sqrt{\frac{3}{2}}\frac{\|D - \frac{1}{3}\mathrm{tr}(D)I\|_F}{\mathrm{tr}(D)}$$

and *fractional anisotropy*

$$a_f = \sqrt{\frac{3}{2}}\frac{\|D - \frac{1}{3}\mathrm{tr}(D)I\|_F}{\|D\|_F}$$

indices, derived by Basser and Pierpaoli [7], have been proposed. Another anisotropy index that is based on both eigenvalues and eigenvectors and takes also

the local neighborhood of a diffusion tensor in a uniform grid into account, the so-called *lattice index*, was introduced by Pierpaoli et al. [160]. Volume ratio, lattice index, relative anisotropy, and fractional anisotropy, however, account only for the overall anisotropy of the tensor, but do not convey information about the actual diffusion behavior, like linear or planar diffusion.

Note that although most of these measures require the tensor to be positive definite they can also provide meaningful results for indefinite tensors if the absolute values of the eigenvalues are used instead of the actual eigenvalues.

Besides scalar contractions also vector valued properties can be derived from a tensor field. For instance, there are the three eigenvector fields corresponding to the major and the two minor eigenvalues. Visualizing the eigenvector fields using streamline integration leads to the concept of *tensor lines* [40] which are in every point tangent to a single principal component of the tensor field. As an extension *hyperstreamlines* [36, 37] as well as *hyperstreamsurfaces* [88] have been defined that utilize the remaining two eigenvectors to define the cross section of a streamtube or a surface respectively. In contrast to point glyphs, such as ellipsoids or the quadric glyphs proposed in Section 6.3.6, scalar contractions and streamline integration-based methods provide a continuous view of the field. Therefore, a combination of the glyph-based method with global visualization techniques such as volume ray casting as has been discussed is beneficial.

6.5 Applications

6.5.1 Diffusion Tensor MRI Data

Diffusion tensor fields obtained by DT-MRI are the prime source of tensor data in visualization today. The visualization of those fields has become a valuable tool in understanding the anatomy and pathology of the brain.

As a first example Figure 6.9 shows an example of a diffusion tensor MRI data set of a human brain[VI] visualized using the ellipsoidal glyphs introduced above. The original data set is regularly sampled on a $148 \times 190 \times 160$ Cartesian grid. It contains about 1.4 million tensor samples that have meaningful values. The remaining data elements either lie outside the brain volume and therefore are not of interest, such as air, skin, bone, and cerebral fluid surrounding the actual brain tissue or have otherwise a low reliability, for example in regions of high noise. Of these tensors only approximately 9000 corresponding to tensors of highly linear diffusion, i.e. $c_l > 0.9$, are shown using ellipsoidal glyphs. They correspond to

[VI] Brain data set courtesy of Gordon Kindlmann at the Scientific Computing and Imaging Institute, University of Utah, and Andrew Alexander, W. M. Keck Laboratory for Functional Brain Imaging and Behavior, University of Wisconsin-Madison.

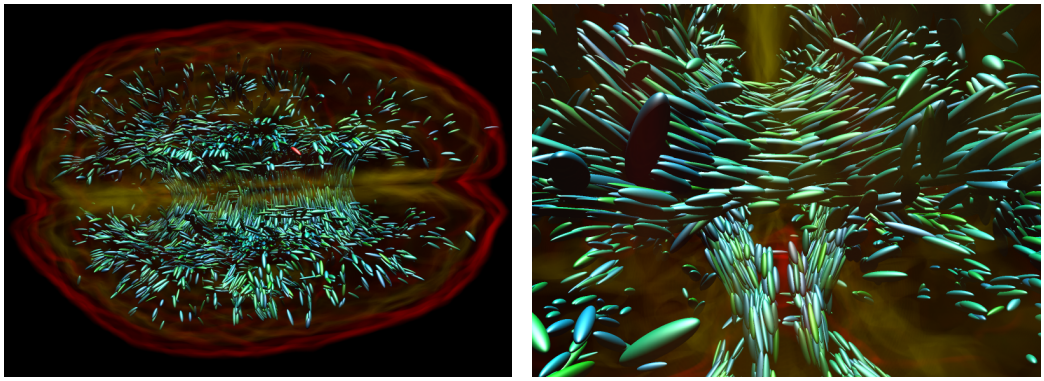


Figure 6.9: Visualization of a diffusion tensor MRI data set of the human brain with a combination of glyphs and volume rendering. On a NVIDIA GeForce 7800GTX, these images can be rendered at about 40 frames per second on a 512^2 viewport. 230 frames per second can be achieved by rendering only the glyphs.

regions that with a high likelihood are expected to contain white matter neuronal pathway structures. These white matter tracts are of special interest in neurosurgery. Additional color-coding was applied to the glyph surfaces according to the fractional anisotropy of the diffusion tensors.

In order to provide additional context of the spatial relationships, the glyph visualization has been combined with a semi-transparent volume ray casting of the corresponding relative anisotropy field. The image to the right shows a detailed view of a central part of the brain, known as the *corpus callosum*. This part is especially rich in nerve fibers since it connects the left and the right hemisphere of the brain.

In order to reduce visual alignment artifacts caused by the regular sampling of the tensor data and to optimize the space utilization by the glyphs a resampling on a hexagonal close-packed 3D lattice [220] has been used. Due to the dense packing of the glyphs this approach improves their spatial distribution. However, it is still not optimal for glyphs other than equally sized spheres since it does not account for the different sizes and shapes of the individual glyphs. Thus, to avoid overlapping there is still much space wasted. A tight packing of glyphs, however, can emphasize the continuity of the underlying field. Various glyph placement strategies that try to optimize the spatial distribution of the glyphs have been described in the literature. However, such glyph packing techniques are beyond the scope of this thesis. But it may be noted that glyph packing techniques based on particle systems utilizing inter-particle force models [237, 97, 81] are promising candidates for GPU-based realization.

6.5.2 Strain and Shear Visualization in CFD

Understanding shear and strain in a fluid is important for understanding the formation and evolution of vortices [121]. Boundary friction and internal friction caused by the viscosity of the fluid give rise to the formation of shear layers. In particular, shear layers play an important role, e.g., in the formation of secondary vortices, in wakes [162], or in the boundary layer of jets [182].

The shear can be derived from the flow's velocity field by decomposing the velocity gradient tensor $J = (j_{ik}) = \nabla \boldsymbol{v}$ of the flow into a symmetric and an antisymmetric part according to Equation (6.1) [55]:

$$\nabla \boldsymbol{v} = \frac{1}{2}(j_{ik} + j_{ki}) + \frac{1}{2}(j_{ik} - j_{ki}) = S + \Omega.$$

This yields the symmetric *strain-rate tensor* S that describes the deformation of an infinitesimal fluid volume element and the antisymmetric *vorticity tensor* Ω that describes the local rigid body rotation of the fluid elements. S can be further divided into *normal strain* and shear strain components. The normal strain described by the diagonal elements of the tensor is related to compression and extension, i.e. changes of the fluid element's volume. In case of an incompressible fluid, i.e. $\nabla \cdot \boldsymbol{v} = 0$, these elements sum to zero. The remaining off-diagonal elements characterize shear strain caused by tangential stress acting on the fluid body.

Shear is typically visualized by scalar contraction of the thus derived tensor field and employing scalar visualization techniques. An isosurface or direct volume visualization of such a scalar field, however, can only convey the amount and distribution of shear deformation but does not account for principal shear directions and strain distribution. Therefore, a combination of strain-rate tensor visualization using tensor glyphs with semi-transparent volume rendering or isosurface visualizations of additionally available or derived scalar fields, is proposed. In fact, both visualization techniques benefit from this combination. The tensor glyphs convey directional information of the strain-rate field while the scalar field provides additional context and spatial cues for the glyphs. Since, in contrast to the diffusion tensor data discussed in the previous example, the strain-rate tensors are in general not positive definite the glyphs proposed in Section 6.3.6 have to be used.

In the literature, two different scalar measures for the visualization of shear magnitude in a flow field can be found. Meyer [146] uses the second invariant I_2 of the strain-rate tensor while Haimes and Kenwright [72] use the second invariant of the strain-rate deviator tensor $S^* = S - \frac{1}{3}\mathrm{tr}(S)I$. Note that for incompressible flow the trace of the strain-rate tensor vanishes and, thus, $S^* = S$, which in turn means that for incompressible flow these two definitions of a stress scalar describe the same property.

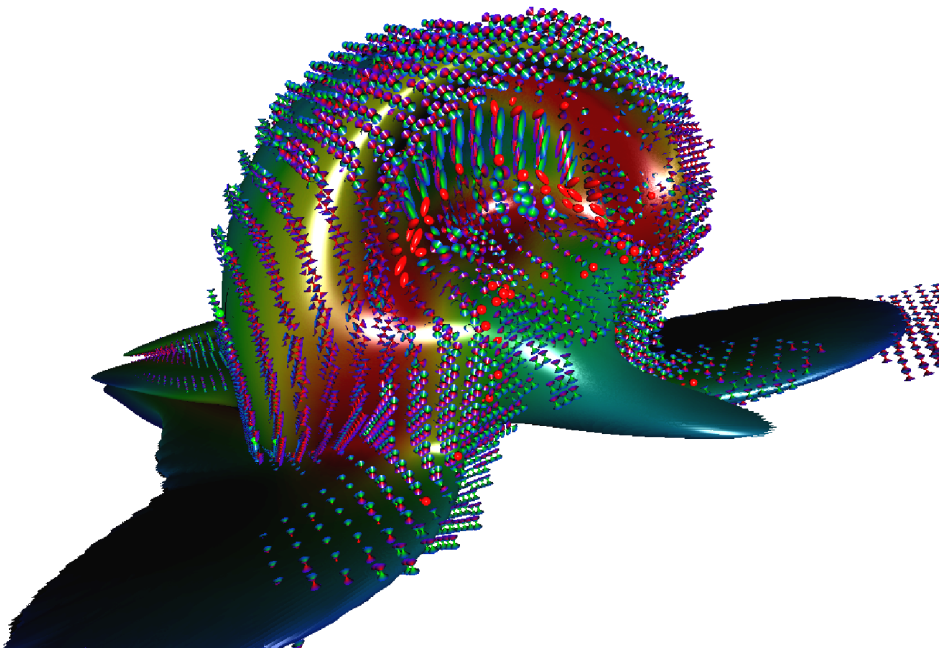


Figure 6.10: Visualization of the shear layer enclosing a vortex using an isosurface of the second strain tensor invariant and quadric tensor glyphs.

An example of an I_2 isosurface visualizing the shear layers forming at fluid boundaries and surrounding a vortex is shown in Figure 6.10. Tensor glyphs are used to depict the local strain properties in the vicinity of the shear layer.

However, using the second invariant of either the strain-rate tensor or the strain-rate deviator as a measure of shear magnitude is kind of an *ad hoc* approach and lacks a real physical motivation. Meyer as well as Haimes and Kenwright chose the second invariant solely since it is a scalar quantity and for its rotational invariance [146, 72]. Therefore, we propose a physically motivated measure. The basic idea is to use the root-mean-square of the shear strain magnitudes as a measure for the average shear stress.

First, we need a number of additional definitions that can be found in any text book on continuums mechanics, e.g., [55]. For a given strain-rate tensor S in point $\mathbf{P}$ the *strain-rate vector* σ acting on an area normal to an arbitrary (normalized) vector $\mathbf{n}$ is given by

$$\sigma = S\mathbf{n}. \tag{6.34}$$

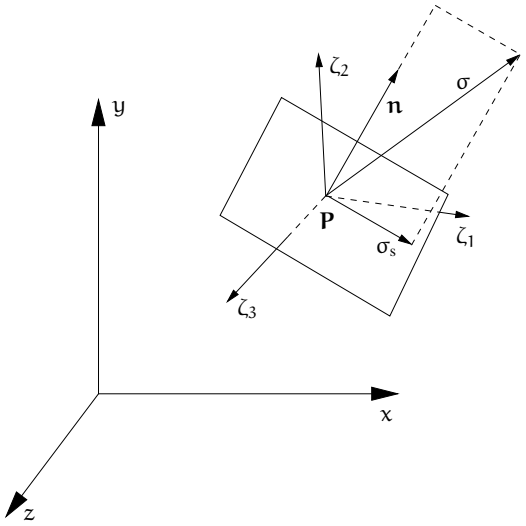


Figure 6.11: Decomposition of the strain-rate vector.

Consequently, the *normal strain* acting in the direction of $\mathbf{n}$ can be computed as

$$N = \mathbf{n}^T S \mathbf{n}. \tag{6.35}$$

Then, the *shear strain* acting tangential to the plane perpendicular to $\mathbf{n}$ is given by

$$\sigma_s = \sigma - N\mathbf{n} = S\mathbf{n} - (\mathbf{n}^T S\mathbf{n})\mathbf{n}. \tag{6.36}$$

Accordingly,

$$|\sigma_s| = |S\mathbf{n} - (\mathbf{n}^T S\mathbf{n})\mathbf{n}| = \sqrt{\mathbf{n}^T S^2 \mathbf{n} - (\mathbf{n}^T S\mathbf{n})^2} \tag{6.37}$$

defines the magnitude of shear strain acting perpendicular to $\mathbf{n}$. Figure 6.11 illustrates the relationship between those quantities.

We can thus define a measure for the mean shear strain acting in $\mathbf{P}$ as the root-mean-square of the shear strain magnitude of all possible shear directions by integrating $|\sigma_s|^2$ over the unit sphere:

$$\begin{aligned} \sigma_m &= \sqrt{\frac{1}{4\pi} \int\limits_{\|\mathbf{n}\|=1} |\sigma_s|^2 d\mathbf{n}} \\ &= \sqrt{\frac{1}{4\pi} \int\limits_{\|\mathbf{n}\|=1} \mathbf{n}^T S^2 \mathbf{n} - (\mathbf{n}^T S\mathbf{n})^2 d\mathbf{n}}\,. \end{aligned} \tag{6.38}$$

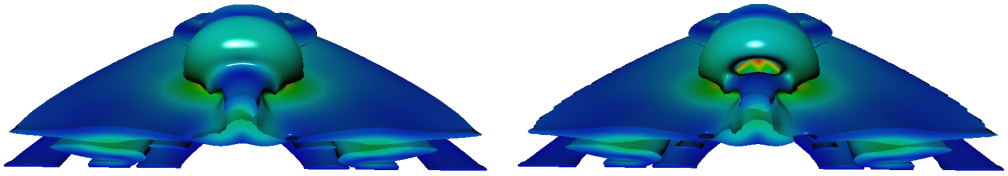


Figure 6.12: Visualization of the shear layer enveloping a vortex using an iso-surface of the proposed physically-motivated shear strain measure (left) and the method by Meyer (right). Additionally, velocity magnitude is mapped on the surface.

Which fortunately evaluates to the closed form

$$\sigma_m = \frac{\sqrt{\pi}}{8}\sqrt{\begin{array}{l}7s_{00}^2 + 7s_{11}^2 + 8s_{22}^2 + 20s_{01}^2 + 32s_{02}^2 + 32s_{12}^2 \\ - 6s_{00}s_{11} - 8s_{00}s_{22} - 8s_{11}s_{22}\end{array}} \tag{6.39}$$

that is in computational complexity similar to I_2 but at the same time physically motivated. Furthermore, in contrast to the I_2 criterion, σ_m is guaranteed to be positive and directly proportional to the magnitude of shear strain acting in $\mathbf{P}$.

First results indicate that the qualitative behavior of the proposed shear scalar is similar to the previous methods. A visual comparison of the shear sheets enveloping a hairpin vortex obtained by the method of Meyer and the proposed approach is shown in Figure 6.12.[VII] However, an in-depth analysis of the proposed shear scalar and a quantitative comparison against the methods by Meyer and Haimes and Kenwright, including a correlation analysis, remains future work.

6.5.3 Vector Field Illustration

As a last application the illustrative visualization of vector fields, in particular the experimentally motivated visualization of magnetic field lines, based on a physically-based model of ferromagnetic particles will be considered.

Experimental visualization of magnetic field lines is a well-known technique. Probably no student leaves school without having done simple experiments carried out with a permanent magnet and some iron filings or other powdery ferromagnetic materials dispersed onto a glass plate in order to visualize the *invisible* magnetic lines of force by observing the characteristic alignment of the particles

[VII] Data set courtesy IAG, University of Stuttgart

along the field lines. With this simple and yet powerful experiment it is possible to gain much insight into the structure and properties of magnetic fields. So why not also use this method most people are familiar with in a computer graphics tool to illustrate the shape of magnetic fields, for example in education or in popular scientific publications? This could be especially useful, if real experiments are not possible due to a complicate setup or complex fields which cannot be reproduced in a small scale experiment are involved.

In order to reproduce the dynamic behavior of particles of ferromagnetic materials in an applied magnetic field a physically based 3D simulation method is employed [100]. Using a full three-dimensional model allows for the simulation of a wide range of phenomena that can be observed during the alignment process of the particles, such as particles that pile up or are restricted in motion due to their shape. It is important to model several physical properties and laws that are involved when particles line up along field lines to achieve the well-known field line pictures. First, there are the mechanical properties of the particles, for example their size and shape and material properties like friction coefficients or elasticity, that, in coaction with for example gravity or obstacle geometry, influence the motion of the particles and their interaction with adjacent particles. As large numbers of finite-sized particles have to be used in order to produce illustrations that resemble the appearance of real world experiments, typical rigid body simulation approaches known from computer animation do not suffice. Instead, methods from an area of computational physics that deal with many-particle systems, for example granular media or particle flows, have to be used.

The actual simulation of the particle dynamics is based on an extremely simplified, but still physically-based model for the magnetostatic and mechanical interactions of the particles. This includes the magnetization of the particles and forces and torques acting on the particles due to the magnetic fields—externally applied as well as induced by the magnetization of the particles—they are exposed to. For several reasons ellipsoidal particles have been chosen. Most importantly they provide a reasonable good approximation to real particle shapes, such as those of iron filings, and furthermore, the ellipsoidal shape exhibits certain magnetic properties that simplify computation of the particle magnetization [156].

Since the evaluation of the long-range magnetic forces as well as the global energy-minimization necessary for computating the particles' magnetization is computationally expensive, simulating systems consisting of several ten thousand particles is only feasible when high performance computers like large shared memory machines or compute clusters are used. A more detailed description of the simulation process and the parallel computation on a cluster computer necessary for large numbers of particles can be found in the original publication [100].

Interactive visualization of this huge amount of scattered particle data requires a fast method to render very large numbers of ellipsoidal particles. Furthermore,

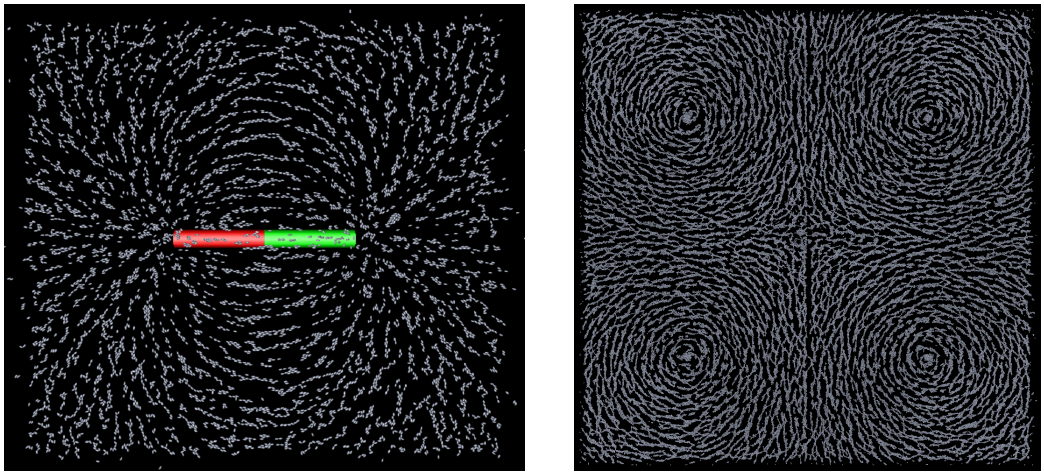


Figure 6.13: Two examples of magnetic field illustrations. On the left: the field of a simple bar magnet illustrated using 6000 particles. On the right: magnetic field lines formed by four current carrying carrying conductors illustrated using 100,000 simulated particles.

since the simulation results are time-dependent, the data has to be streamed from main memory to the GPU for every rendered frame. Thus, the point-based approach introduced in this chapter is very well suited for this task.

Figure 6.13 shows still images from two showcase simulations. In the first example a rather small amount of only 6000 particles was used to illustrate the field of a simple bar magnet positioned a short distance below a thin magnetic non-shielding plate, such as glass or paper. Nevertheless, the general structure of the underlying field is clearly visible, even for this small number of particles. However, for more complex fields and better visual quality, especially for images that resemble the pictures of real-world experiments, more particles should be used. Using 20,000 to 100,000 particles in the simulation has been found to provide reasonably good results. As a second example a time-step of a simulation of 100,000 ferromagnetic particles in the field excited by four current carrying conductors is shown.

As a third example, Figure 6.14 shows a variation of another typical example often used in introductory texts on electromagnetism. Here the field of two solenoids—magnetizing coils with air cores—is illustrated. In this example the dynamic behavior of 30,000 particles has been simulated over 50,000 timesteps corresponding to 150ms overall simulation time. However, only 5000 timesteps have been saved during the simulation, which, nevertheless, results in more than 2.2GB of raw simulation data that can be interactively explored with the point-

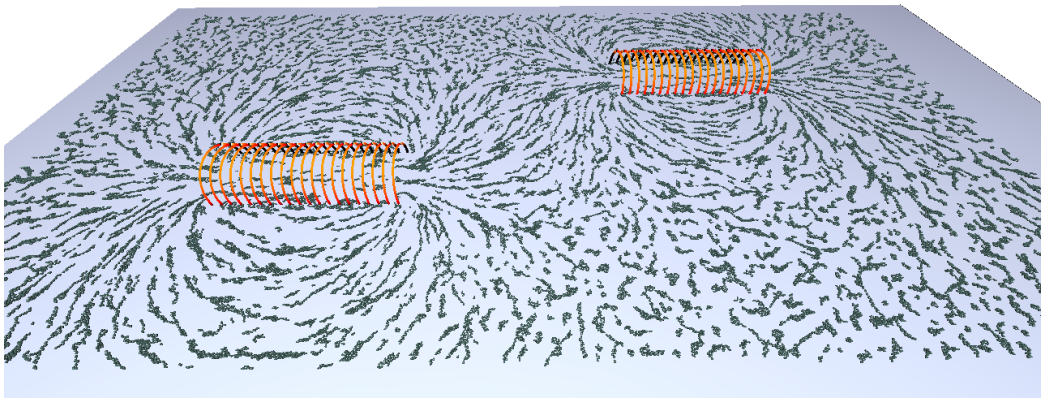


Figure 6.14: Another example of magnetic field illustration. Magnetic field lines of two solenoids illustrated by 30,000 ferromagnetic particles aggregated along field lines.

based visualization approach. Additional geometry is used in Figure 6.14 to depict the nonmagnetic, nonshielding plate, the particles are strewn on, and the two coils that induce the magnetic field.

6.6 Conclusion

In this chapter a grid independent visualization technique for tensor field data has been presented. The proposed point-based approach does not depend on tesselated geometry but reduces the necessary geometric data to the bare minimum. This minimizes the amount of data transmitted from the CPU to the GPU and reduces the load on the graphics bus. Thus, it enables the possibility for streaming time-dependent data for interactive visualization, so that large tensor fields can be interactively explored.

The widely used tensor ellipsoid representation has been extended for the visualization of indefinite tensor fields. Furthermore, the combination of the glyph rendering with continuous scalar field visualization techniques allows to reduce visual clutter and to provide additional context information.

Several applications ranging from medical diffusion tensor visualization to vector field illustration have been shown to demonstrate the effectiveness and flexibility of the proposed point-based ray casting approach. Another evidence for the generality of the point-based method is that the same approach has been also successfully employed for visualizing molecular structures [170, 194].

Chapter 7

Topological Analysis of Vector Fields

In many cases straightforward visualization of a data field is not possible, either because the data is too large for interactive visualization due to limited hardware resources or the shear amount of information presented by a global visualization approach and the corresponding visual complexity of the visualization would put too much burden onto the user's perception when asked to identify important patterns or structures in the data. In this case it is common to employ feature extraction techniques in order to create a compact representation of the data that captures the important characteristics of the underlying field in an abstract representation.

In the previous chapter abstract representations of local flow field features have been discussed. However, using a very large number of glyphs to achieve a comprehensive overview of the whole field poses not only a high rendering overhead but comes at the cost of extensive visual clutter. This is particularly problematic for three-dimensional fields. Here occlusion is a major problem.

On the other hand one is often only interested in a qualitative overview of the global structure of the data in the form of a high-level abstract representation, which provides an abstract depiction of the field in a strongly condensed form. In case of vector fields the topological analysis is an established technique to show the essential properties and the qualitative structure of the field. Based on vector field singularities and a set of feature lines and surfaces—the so-called *topological skeleton*—, the flow domain is partitioned into subregions of qualitative equivalent behavior.

In this chapter a new approach for the topological analysis of noisy vector fields is presented that has been published in similar form in [101].

7.1 Vector Field Topology

The definition of flow topology and the topological skeleton originates from the analysis of dynamic systems [204]. For a given vector field

$$\boldsymbol{v}:\mathsf{D} \rightarrow \mathbb{R}^3, \quad \boldsymbol{x} \mapsto \boldsymbol{v}(\boldsymbol{x}) \tag{7.1}$$

on a domain $\mathsf{D} \subseteq \mathbb{R}^3$ the *flow* of the field is defined as a map

$$\phi:\mathsf{D} \times \mathsf{T} \rightarrow \mathbb{R}^3, \quad (\boldsymbol{x}, t) \mapsto \phi(\boldsymbol{x}, t) \tag{7.2}$$

with $\mathsf{T} \subseteq \mathbb{R}$, where $\frac{d}{dt}\phi(\boldsymbol{x}, t) = \boldsymbol{v}(\phi(\boldsymbol{x}, t))$ for all $(\boldsymbol{x}, t) \in \mathsf{D} \times \mathsf{T}$. Which in turn leads to the definition of *integral curves* of the vector field, i.e. curves whose tangent vectors at each point along the curve is the vector field itself at that point and which are uniquely defined for each point $\boldsymbol{x}_0 \in \mathsf{D}$ by $\frac{d}{dt}\phi(\boldsymbol{x}_0, t) = \boldsymbol{v}(\phi(\boldsymbol{x}_0, t))$ and $\phi(\boldsymbol{x}_0, 0) = \boldsymbol{x}_0$. In the case of a stationary vector field these integral curves are also called *streamlines* of the field. The set of integral curves is called the *phase portrait* of the flow and the qualitative structure of the phase portrait, i.e. the behavior of the streamlines, is called the *topology* of the field.

The topology of the field is characterized by its singularities or *critical points*, i.e. positions where the vector field vanishes: $\|\boldsymbol{v}(\boldsymbol{x})\| = 0$. Critical points play a special role in a vector field since there cannot exist a uniquely defined streamline in such a point. Loosely speaking, these are the only points in the field where streamlines can (asymptotically) meet. The nature of a critical point can be classified according to the behavior of the streamlines in its vicinity. This can be formalized by studying the first-order approximation

$$\boldsymbol{v}(\boldsymbol{x}) \approx \boldsymbol{v}(\boldsymbol{x}_c) + \boldsymbol{v}_{\boldsymbol{x}}(\boldsymbol{x}_c)(\boldsymbol{x} - \boldsymbol{x}_c) = \boldsymbol{v}_{\boldsymbol{x}}(\boldsymbol{x}_c)(\boldsymbol{x} - \boldsymbol{x}_c) \tag{7.3}$$

of the field at the location of the critical point $\boldsymbol{x}_c$. Eigenanalysis of the Jacobian $\boldsymbol{v}_{\boldsymbol{x}}$ at $\boldsymbol{x}_c$ leads to the classification of linear or first-order critical points [204]. There are two possibilities: either all three eigenvalues are real or one is real and the other two form a pair of complex conjugates. This leads to four basic types of first-order critical points in a 3D flow: nodes (three real eigenvalues of the same sign), saddles (three real eigenvalues of different signs), spirals (one real and a complex conjugate pair of eigenvalues where the real part of the complex eigenvalues has the same sign as the single real eigenvalue), and spiral saddles (one real and a complex conjugate pair of eigenvalues where the real part of the complex eigenvalues has a sign different to the single real eigenvalue). The sign of the real parts of the eigenvalues further indicate, whether the critical point repels—positive sign—or attracts—negative sign—streamlines in its vicinity. Note that this requires the Jacobian to have full rank, i.e. $\det(\boldsymbol{v}_{\boldsymbol{x}}) \neq 0$ or equivalently none

of its eigenvalues being zero. If this is not the case, higher-order terms of the Taylor expansion have to be taken into account and the corresponding points are called nonlinear or higher-order critical points. In the following, however, only first-order critical points will be considered.

A special role play streamlines that originate from saddle points. These so-called *separatrices* partition the flow domain into regions of qualitative similar behavior. Since each 3D saddle point consists of an inflow/outflow plane and an outflow/inflow direction, which are indicated by the dimensionality of the eigenspaces associated with the eigenvalues, these separatrices form either streamsurfaces or streamlines that represent the topological skeleton of the flow.

There is a huge body of literature dealing with the extraction and visualization of vector field topology. Since their first introduction to the context of visualization of two-dimensional vector field data by Helman and Hesselink [77] topology-based methods have been established as one of the basic tools for flow analysis and were soon generalized to three-dimensional fields, as well [78, 60].

Since then, many improvements and extensions have been published. The application of results from geometric calculus and Clifford algebra regarding vector field indices made it possible to detect and visualize also higher-order critical points and nonlinear vector field topology. This has been investigated in detail by Scheuermann et al. [185, 186, 187] for two-dimensional flows and by Mann and Rockwood [138] for three-dimensional vector fields.

As visualizations of the topology of three-dimensional flow fields typically involve a large number of separating surfaces they suffer from occlusion problems and accordingly tend to get visually cluttered. To deal with that problem, Theisel et al. introduced the concept of saddle connectors [206] and boundary switch connectors [219] as a method for the simplified visualization of the topological skeleton of complex three-dimensional vector fields. However, the visual complexity, even of these simplified visualizations, depends heavily on the number of critical points involved.

In order to deal with large numbers of critical points and the resulting complexity of the topological skeleton, the multi-scale nature of the data has to be addressed in one way or the other. The simplest approach is to use a low pass filter to suppress small-scale features and noise in the data before extraction of the critical points. However, this approach cannot guarantee the invariance of the position and classification of critical points in the data set. Therefore, a number of topological simplification methods have been proposed. Examples are the work by de Leeuw and van Liere [34] or by Tricoche et al. [208]. In contrast to the approach that will be described in the following, however, these methods often depend on the availability of the complete topological skeleton, not merely the critical points.

Besides direct visualization of the topological skeleton topological features

have been also used in a number of applications to guide other visualization techniques. One example is streamline seeding guided by topological information [212]. In this paper different types of seed templates are defined based on the classification of critical points and used for automatic seeding of streamlines. This way, all important features of the flow field are captured by the streamlines without requiring the generation of large numbers of unnecessary lines as would be the case by uniform seeding.

7.2 Detection of Critical Points

In the following only vector field singularities or critical points that are spatial isolated zeros of the vector field with nonsingular Jacobian matrix, i.e. first-order critical points, will be considered. This poses in general no problem, as line or surface singularities always correspond to degenerate critical points, i.e. points where the Jacobian of the field does not have full rank. Obviously, this also excludes higher-order critical points that may be present in nonlinear vector fields [185].

The detection of vector field singularities is in general a numerically ill-posed and challenging problem. Noise and numerical inaccuracies will lead to false positives and actual singularities might be missed due to deficiencies of the employed interpolation scheme or insufficient sampling of the data field. As for all topology-based methods finding a complete seed set of critical points is a crucial part of the algorithm.

Two different approaches for the detection of critical points on a regular three-dimensional grid have been implemented and compared; namely the linearization of the vector field using a tetrahedral decomposition of the grid and a method based on the computation of winding numbers using geometric calculus. Both methods have certain strengths and weaknesses.

Decomposing the grid cells into tetrahedra corresponds to effectively linearizing the vector field, thus higher-order critical points cannot be detected directly by this method. Such points either are missed or show up as pairs of neighboring first-order critical points [185]. Although for each grid cell multiple tetrahedra have to be processed, the actual computation per tetrahedron boils down to solving a system of linear equations for the barycentric coordinates of the critical point's position inside the cell. Therefore, this method is still the simplest and fasted way to compute vector field singularities as long as higher-order critical points can be neglected.

Second, the method described by Mann and Rockwood [138] based on computing the index of a critical point using winding numbers has been investigated. At the moment this is regarded as the most general method, since it is not restricted to detecting first-order critical points but can theoretically detect critical points

of arbitrary index. Using results from geometric calculus Mann and Rockwood determine the index of a critical point by computing an integral over a closed surface. The result of this computation is always an integer value that is either zero if no critical point is enclosed by the surface or equals the index of the critical point enclosed by the surface. Unfortunately, there are also some problems with this approach. First of all, the value of the integral in general may not reflect the actual situation. Since the indices of multiple singularities enclosed by the surface add up in the result of the integration, an index of zero does not necessarily signify that no critical point is enclosed. The same applies for values not equal to zero. The problem is how to choose the appropriate spatial resolution for the enclosing manifold. At first sight one cell or the surface defined by its six faces seems to be a good choice, since this defines the maximum resolution of our data. But, as can be easily seen, a single cell is not sufficient to identify, for example a dipole singularity in a magnetic field, at least if bilinear interpolation is used for sampling the cell faces. Thus, expensive higher-order interpolation schemes have to be used. But even when using tricubic interpolation, we experienced singularities being missed. Furthermore, in order to achieve a reasonable accuracy for the integration, i.e. indices that are close to integer values, the sampling grid on the cell faces has to be sufficiently fine, which makes the computation even more expensive. Therefore, this method is not feasible for real data sets.

7.3 Scale-Space Techniques

Since typical data sets originating from simulations or real world measurements contain structures of different sizes or scales and different sets of features can be observed on certain ranges of scale, the notion of scale is an important concept in the analysis of flow data. Therefore, an automatic or semi-automatic analysis tool for flow topology has to accommodate the inherent multi-scale nature of the underlying data.

Furthermore, many flow features, for example vortices, can be observed on multiple scales or resolutions of the data while many features that can be only detected at fine scales are essentially artifacts of the employed interpolation scheme or originate from noise in the data. Thus, a scale-aware technique can help to identify the overall structure of a given flow field and allows to distinguish the global structure from local effects, such as noise, and discretization or interpolation artifacts.

As in general the scales of the features are a priori not known, it seems to be reasonable to represent the data at multiple scales, successively eliminating fine scale flow field structures. Nevertheless, in certain situations the scientist analyzing the data may be in possession of information that allows to limit the

parameter space to certain resolutions. The concept of scale-space provides a well established framework to cope with these problems in a well defined way.

Scale-space techniques have become very popular in image processing and computer vision since they allow for the integrated analysis of the image structure. Scale-aware feature extraction techniques for scalar valued data have a long tradition. They enable one to distinguish between important features, such as edges, and small-scale features, such as numerical artifacts or noise. By reason of this, scale-space techniques have been successfully applied to the automatic analysis of images in computer vision. It was in the context of pattern recognition when the concept of scale of features and the scale-space paradigm emerged first [86]. Starting with the work of Witkin [231] and Koenderink [106] the concept of Gaussian scale-space representations has gained much attention in the image processing literature. The analysis of the so-called *deep structure* of images by means of an investigation of a multi-scale image representation, has become a valuable tool for feature detection and extraction in scalar images [130]. Of special interest in the context of this chapter is the work by Lindeberg [129] and by Florack and Kuijper [53, 116] on the scale-space behavior of critical points in scalar fields, since the approach that will be described in the following is closely related to their work on critical curves, i.e. the trajectories of singularities in the gradient field of a scale-space image. However, we will disregard their extensive work on bifurcations and degenerate singularities based on the framework of catastrophe theory [115].

In this chapter a first attempt of a multi-scale topological analysis of flow data based on Gaussian scale-space hierarchies of three-dimensional vector fields will be presented. The main focus is put on the tracking of important topological features, i.e. critical points, over multiple spatial scales in order to assess the importance of a critical point to the overall behavior of the underlying flow field or, in other words, to distinguish between local and global structures and numerical or noise-induced artifacts. This procedure is based on the assumption that fine scale structures and noise will be gradually eliminated on coarser scales while the dominating large scale flow structures will be persistent over multiple or all scales of the data set. Thus, it is assumed that the scale length of the path or the number of scales over that a critical point can be tracked corresponds to the importance of this singularity to the overall topology of the flow.

7.3.1 Gaussian Scale-Space

Introducing the scale parameter $\tau > 0$, the scale-space representation

$$v : D \times \mathbb{R}^+ \to \mathbb{R}^3, \quad (\mathbf{x}, \tau) \mapsto v(\mathbf{x}, \tau) \tag{7.4}$$

of the vector field $\boldsymbol{v}$ with $\lim_{\tau\to 0} v(\boldsymbol{x},\tau) = \boldsymbol{v}(\boldsymbol{x})$ defines an embedding of the original vector field in a one-parameter family of derived vector fields, resulting in a hierarchy consisting of subsequently simplified or smoothed versions of the original data. Thus, τ defines the scale axis of the four-dimensional scale space.

One possibility, and in fact the most often used one, to generate such a one-parameter family of vector field representations is the linear or Gaussian scale space. In this case the smoothing of the data is accomplished by the convolution $v(\boldsymbol{x},\tau) = \boldsymbol{v}(\boldsymbol{x}) * g(\boldsymbol{x},\tau)$ of the original signal with Gaussian filter kernels

$$g(\boldsymbol{x},\tau) = \frac{1}{(2\pi\tau)^{3/2}} e^{-\frac{\|\boldsymbol{x}\|^2}{2\tau}} \tag{7.5}$$

of increasing standard deviation $\sigma = \sqrt{\tau}$, which is equivalent to the solution of the linear diffusion or heat equation [106]

$$\partial_\tau v(\boldsymbol{x},\tau) = \frac{1}{2}\Delta v(\boldsymbol{x},\tau)\ , \tag{7.6}$$

with initial condition

$$v(\boldsymbol{x},0) = \boldsymbol{v}(\boldsymbol{x})\ .$$

Here $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ denotes the spatial Laplacian operator.

Hence, for a vector field given on a discrete grid a Gaussian scale-space representation can be computed in at least two ways, by repeatedly filtering the data with sampled Gaussians of increasing variance and accordingly width or by solving the diffusion Equation (7.6). Although the second involves the solution of a partial differential equation, for large τ this methods becomes increasingly efficient compared to repeated convolution with sampled Gaussians.

For the computation of the scale-space representation both methods have been implemented: the repeated convolution with separated Gaussians of increasing variance and accordingly increasing support and the iterative solution of the linear diffusion equation using a finite difference approximation of Equation (7.6) and a Gauss-Seidel solver employing successive over-relaxation.

7.4 Tracking of Critical Points in Scale Space

This section will be concerned with the development of an algorithm for tracking nondegenerate singularities in scale space.

The tracking of features in vector fields in general has attracted the interest of a number of researchers in flow visualization, especially in the case of the analysis of time-dependent vector fields. A general overview of the state of the art

in feature extraction and tracking in flow fields can be found in Post et al. [166]. Particularly concerned with the tracking of vector field singularities as a topological feature in time-dependent two-dimensional and three-dimensional data sets are the works by Tricoche et al. [209, 210], Theisel and Seidel [205], and Garth et al. [57]. Feature tracking in scale-space representations of vector fields has been investigated by Bauer and Peikert [8] for tracking the evolution of vortices in scale and time.

For the approach discussed here the papers by Theisel and Seidel [205] and Bauer and Peikert [8] are the two most relevant. In the first, the concept of feature flow fields is introduced. A vector field derived from the original flow field such that the evolution of the considered features can be described by streamlines in this field. Thus, the feature tracking problem is reduced to simple streamline integration. This technique will be described in detail in Section 7.4.1. In the second, vortex core lines based on the parallel vector operator are investigated. The construction of a linear scale-space representation of a flow field given on a three-dimensional unstructured grid is discussed and an algorithm for the tracking of the vortex core lines in scale space based on a 4D surface extraction technique is presented. To our knowledge, this is the only work on scale-space feature tracking for flow visualization.

A straightforward solution for tracking critical points would be to detect all critical points on all scales and then to try to connect them according to some simple correspondence criteria, such as spatial distance or classification. However, there are a number of problems with this approach. In many situations such simple criteria are not sufficient to decide whether a critical point detected on a coarser scale really corresponds to a spatially close point on the next finer scale. Furthermore, tracking based on feature correspondence depends somehow or other on the definition of certain threshold values, e.g. for the maximum distance of two features considered to be equal. Often it is not possible to specify such values for an unknown data set in advance. Thus, a fully automatic analysis of the data is not possible. In the following two approaches will be described that do not require such thresholds. The first is based on the concept of feature flow fields. The second exploits the explicit knowledge of the scale-space trajectory of a critical point provided by the implicit function theorem. Both approaches reduce the feature tracking problem to a streamline integration problem in a derived vector field, as will be described in the following.

7.4.1 Feature Flow Field Approach

As introduced by Theisel and Seidel [205], a so-called feature flow field of a time-dependent three-dimensional vector field $\mathbf{u}(\mathbf{x}, t) = (u_1(\mathbf{x}, t), u_2(\mathbf{x}, t), u_3(\mathbf{x}, t))^T$ is a four-dimensional vector field $\mathbf{f}(\mathbf{x}, t) = (f_1(\mathbf{x}, t), \ldots, f_4(\mathbf{x}, t))^T$ derived from

$\mathbf{u}(\mathbf{x}, t)$ such that the time evolution of the considered features is described by streamlines in $\mathbf{f}(\mathbf{x}, t)$. In the case of a critical point $\mathbf{x}_0$ this means each point on the streamline in $\mathbf{f}(\mathbf{x}, t)$ starting from $\mathbf{x}_0$ is also a critical point, i.e. the value of $\mathbf{u}(\mathbf{x}, t)$ must not change when traversing the streamline. Assuming a first-order approximation of $\mathbf{u}(\mathbf{x}, t)$ around $\mathbf{x}_0$ this implies that $\mathbf{f}(\mathbf{x}, t) \perp \nabla u_i(\mathbf{x}, t)$ for $i = 1, 2, 3$. When this is applied to the scale-space representation $\nu(\mathbf{x}, t)$ by identifying time t with scale τ, the scale-space feature flow field $\mathbf{F}(\mathbf{x}, \tau)$ is given by

$$\mathbf{F}(\mathbf{x}, \tau) = \begin{pmatrix} -\det(\nu_y(\mathbf{x}, \tau), \nu_z(\mathbf{x}, \tau), \nu_\tau(\mathbf{x}, \tau)) \\ \det(\nu_z(\mathbf{x}, \tau), \nu_\tau(\mathbf{x}, \tau), \nu_x(\mathbf{x}, \tau)) \\ -\det(\nu_\tau(\mathbf{x}, \tau), \nu_x(\mathbf{x}, \tau), \nu_y(\mathbf{x}, \tau)) \\ \det(\nu_x(\mathbf{x}, \tau), \nu_y(\mathbf{x}, \tau), \nu_z(\mathbf{x}, \tau)) \end{pmatrix} , \tag{7.7}$$

where $\nu_x(\mathbf{x}, \tau))$, $\nu_y(\mathbf{x}, \tau)$, and $\nu_z(\mathbf{x}, \tau))$ are given by the columns of the spatial Jacobian $\nu_x(\mathbf{x}, \tau)$ of the scale-space representation.

Thus, the feature tracking problem is reduced to a four-dimensional streamline integration problem starting at an initial set of seed points given by the critical points encountered on the finest scale.

7.4.2 Implicit Function Theorem Approach

In this section an alternative approach for critical point tracking is described that has been used, for example, by Lindeberg [129] for analyzing the behavior of local extrema in images under Gaussian blurring. Similar to the feature flow field approach, the basic idea here is that using explicit knowledge about the actual trajectories of the critical points in scale space can significantly improve the tracking results.

As will be shown in the following, the evolution of a nondegenerate critical point in scale space can be analyzed by means of the general implicit function theorem. Computing the trajectory of a critical point in scale space can be regarded to be equivalent to finding the level set $H : \nu(\mathbf{x}, \tau) = 0$. Then, for a given scale-space critical point $(\mathbf{x}_0, \tau_0) \in H$ with nonsingular matrix $\nu_\mathbf{x}(\mathbf{x}_0, \tau_0)$, i.e. $\det(\nu_\mathbf{x}(\mathbf{x}_0, \tau_0)) \neq 0$, the implicit function theorem states the following: In a local neighborhood of the critical point $\nu(\mathbf{x}, \tau) = 0$ can be solved for $\mathbf{x}$. In other words, there exists a function $\mathbf{h}(\tau)$ with $\mathbf{x}_0 = \mathbf{h}(\tau_0)$ and $\nu(\mathbf{h}(\tau), \tau) = 0$. Hence, the path of a critical point in the four-dimensional scale space is a one-dimensional manifold, i.e. a curve. Although it is not guaranteed that there exists an explicit representation of $\mathbf{h}$, the tangent of the curve in $(\mathbf{x}_0, \tau_0)$ can be always computed as

$$\mathbf{h}'(\tau_0) = -(\nu_\mathbf{x}(\mathbf{x}_0, \tau_0))^{-1} \nu_\tau(\mathbf{x}_0, \tau_0) . \tag{7.8}$$

Repeating this argument, the integration of the path of the singularity through scale space can then be accomplished by solving the following differential equation

$$\partial_\tau \mathbf{h}(\tau) = -(\nu_{\mathbf{x}}(\mathbf{h}(\tau),\tau))^{-1}\nu_\tau(\mathbf{h}(\tau),\tau)\ , \tag{7.9}$$

with initial condition $\mathbf{h}(\tau_0) = \mathbf{x}_0$.

7.4.3 Predictor-Corrector Algorithm

Regardless of the decision whether to use the method based on the feature flow field or the implicit function theorem in both cases a streamline has to be traced for all critical points detected in the original vector field. The methods differ only in that in the first case a four-dimensional streamline integration has to be performed whereas for the second method an integration in three dimensions is sufficient.

The basic algorithm then is as follows: First, all critical points on the finest scale level, i.e. the original data set, are extracted using one of the methods described in Section 7.2. Note that this does not guarantee a complete seed set for the topology of the scale-space representation of the data, but is sufficient to compute the scale-space evolution of the critical points under consideration. However, as mentioned above, finding all critical points in a sampled vector field is a general problem of topological analysis.

Then for each critical point a streamline is traced according to Equations (7.7) or (7.9). Note also that in contrast to the more general case of streamline-based feature tracking in time-dependent flow fields investigated by Theisel and Seidel [205], it is sufficient to do only a forward integration of the streamline here.

In both cases scale-space derivatives have to be computed. This can be either accomplished by precomputing the field ν for a fixed number of scales or by concurrent filtering/diffusion and streamline integration. The first is easy to implement but entails a high memory overhead for storing a whole hierarchy of three-dimensional vector fields. Since streamline integration is only forward in scale, only two scale levels are necessary when the computation of the scale-space representation and the streamline integration are done concurrently.

Furthermore, because we are working in a linear scale space we have explicit knowledge about the derivative with respect to τ. Since $\partial_\tau \nu(\mathbf{x},\tau) = \frac{1}{2}\Delta\nu(\mathbf{x},\tau)$ the derivative with respect to scale can be derived from the second derivatives of $\nu(\mathbf{x},\tau)$ in space. Thus both $\partial_x \nu(\mathbf{x},\tau)$ and $\partial_\tau \nu(\mathbf{x},\tau)$ can be computed by differentiation in the spatial dimension followed by a linear interpolation between two consecutive scale levels.

Last, the described streamline integration process has to be stopped as soon as an annihilation event is reached, as will be discussed in the next section.

Unfortunately, both methods turned out to be numerically unreliable when implemented in this straightforward fashion. Although they work well for simple analytic test data sets, both fail to capture the correct behavior for noisy real world data sets. Even when using a fourth-order Runge-Kutta integration scheme with adaptive step size control for the computation of the streamlines, the computed traces very soon deviate from the actual paths of the critical points in scale space that have been computed for comparison by repeated extraction of critical points on multiple scale levels. The major problem seems to be the accuracy of the computed derivatives. They have to be numerically approximated from the sampled vector field and, although a tricubic interpolation scheme is used, their accuracy, in particular that of the second derivatives, is not very high. Since using even higher interpolation order would give rise to a disproportionately large computational effort this is not an option. Therefore, the scale-space trace integration has been combined with a Newton-Raphson method in a predictor-corrector approach. In each step of the streamline integration a prediction is computed by integrating in the direction indicated by either the feature flow field $\mathbf{F}(\mathbf{x}, \tau)$ or the right hand side of Equation (7.9). Afterwards, this prediction is refined using a typically very small number of Newton-Raphson iterations. Now it turned out that even a simple third-order Runge-Kutta method is sufficient to compute reliable scale-space traces.

7.4.4 The Problem of Bifurcations

The additional degree of freedom introduced by the scale parameter, in fact, provides further topological features. There exist transitions between the different scale levels that cause topological changes. Subtle changes of the scale parameter can cause structural changes in the system's topology. These so-called bifurcations or catastrophes can occur in every dynamic system that depends on a set of varying control parameters, such as scale in our case. Thereby, the local topology changes from one stable state to another via a transient unstable state.

These phenomena have been studied extensively in the theory of dynamic systems [62] and can be described in the framework of catastrophe theory that deals with the problem how critical points of a parameter dependent dynamic system will evolve when the control parameters are continuously changed. No further detail on this topic will be provided here. An extensive account of the behavior of critical points of scalar fields, i.e. singularities of their gradient fields, under Gaussian blurring can be found, for example, in the work by Florack and Kuijper [53, 116].

Many different types of possible bifurcations exist in a general dynamic system [62], but in flow topology one is mostly concerned with three general types of bifurcation events that can occur in the topology of a parameter dependent vec-

tor field: annihilations and creations of pairs of critical points and critical points changing their classification. The last, the so called Hopf bifurcation, describes for example the transition from a sink to a source or vice versa.

Unfortunately, the possibility of creation events in a linear scale-space representation seems to be contradictory to the goal of topology simplification. New (unstable) degenerate critical points can be created during the diffusion process that subsequently will lead to the creation of a new pair of critical points. This is known as a static fold bifurcation. However, since we are only interested in the behavior of the critical points that exist in the original data set, i.e. on the finest scale, this type of bifurcation event can be safely ignored.

Obviously, Equations (7.7) and (7.9) are only valid as long as the Jacobian $\nu_{\mathbf{x}}(\mathbf{x}, \tau)$ is not singular. Therefore, the Jacobian matrix must be monitored during the integration of the scale-space trace in order to capture the bifurcation events and to be able to stop the integration of the trace, accordingly. Note that the Jacobian $\nu_{\mathbf{x}}(\mathbf{x}, \tau)$ is a smooth function of the scale parameter τ and, accordingly, its determinant is also a smooth function of τ. Thus, a sign change of the Jacobian determinant signifies a bifurcation. There are now two possibilities for the kind of event that has occurred. The critical point is either passing through the unstable state of a Hopf-type bifurcation or a fold bifurcation point has been reached and the traced critical point is annihilated. In the second case, the condition is sufficient to detect the event since in an annihilation event always two critical points with opposite sign of their Jacobian determinant are involved. But although a wide range of possible Hopf-type bifurcations can be also detected that way, some of them are hard to detect since the Jacobian determinant does not change its sign but instead passes through a second-order zero of the characteristic polynomial. In this case it is very unlikely, that the event would be even noticed. Nevertheless, this problem could be solved by computing the eigenvalues of the Jacobian. On the other hand, this poses a large computational overhead that is in fact completely unnecessary. Since we are interested in classifying the critical points according to their scale-space lifetime Hopf bifurcations can be safely neglected because only the type of the critical point is changed not the fact of its existence. That means, the problem is not whether a Hopf bifurcation has been missed, but in the case a zero determinant is encountered if it is an annihilation event or not. Therefore, if $\det(\nu_{\mathbf{x}}(\mathbf{x}, \tau))$ is zero we have to check if it is a real zero crossing by going one step further in the direction of the streamline before terminating the streamline integration.

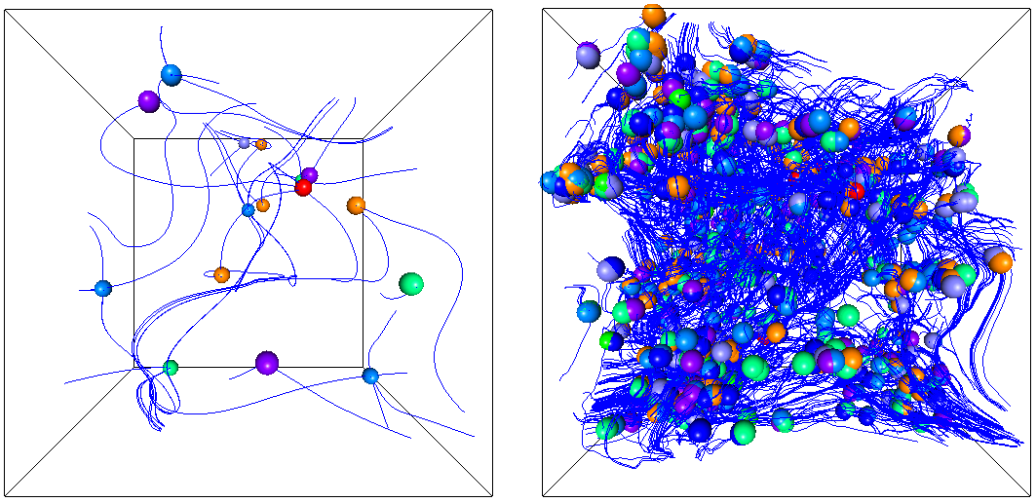


Figure 7.1: A random generated test data set. Critical points are color coded according to their classification. Foci are shown in red and green hues, while saddles are colored in blue tones. Left: Original data. Right: Data with noise added.

7.5 Evaluation

In this section the above described scale-space tracking approach is applied both to a generated test data set as well as to real world flow data sets.

The first data set is an artificial test data set. It was created by resampling a random generated 10^3 vector field to a 50^3 grid using tricubic filtering. In this data set 16 first-order critical points (3 saddles, 3 focus saddles and 10 foci) have been detected. The original 16 critical points in combination with streamlines seeded in their vicinity are shown in the left image of Figure 7.1. The same data set after adding some noise is shown in the right image of Figure 7.1. In this case, normal-distributed noise was added to 20% of the vector components of the field, which leads to a rather low signal-to-noise ratio of approximately 11dB for this data set.

The number of critical points that can be detected now is 1307 and the topology is much too complex to be of any practical use. Applying the proposed scale-space tracking scheme enables us to distinguish between critical points that have been solely introduced due to noise and critical points that represent the dominating flow behavior.

For this test, the points were traced using the scheme derived in Section 7.4.2 over the scales from $\tau = 0$ to $\tau = 1.5$. The computation took approximately 70 second on a machine equipped with an AMD Opteron 2.0GHz processor and 8GB of main memory; 8.5 seconds were spent for computing the scale-space representation and 61 second for the actual streamline integration. The numbers for

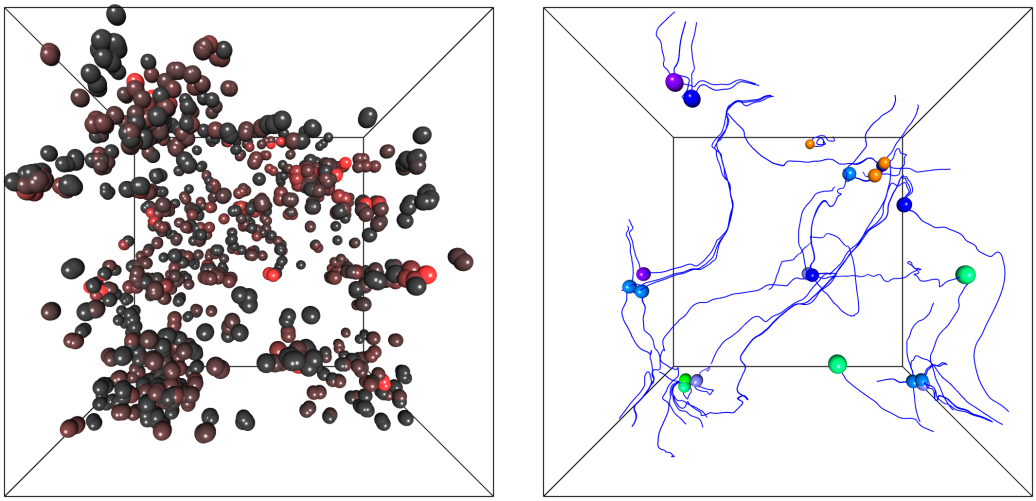


Figure 7.2: The same test data set as shown in Figure 7.1. Left: Scale-space lifetime of the critical points in the interval $\tau = 0 \ldots 1.5$ computed by our algorithm. Bright red color indicates stable critical points, while dark colored points are very short-lived. Right: Critical points filtered by their lifetime. Only points that persist at $\tau_{min} = 1.0$ are shown.

the feature flow field based method of Section 7.4.1 are comparable and identical results are produced for both methods. Figure 7.2 shows the result of this computation. A gray to red color ramp is applied to indicate the scale-space lifetime of the critical points in the left image. Points shown in dark gray are very short lived while light red colored points could be traced over the whole scale interval. The actual scale-space traces of the critical points are not shown in the images, since they do not provide much of additional information. They may only serve to identify pairs of critical points that participate in annihilation events.

Last, in the right image of Figure 7.2 the result of filtering the points according to their scale-space lifetime is shown. Only points that have a lifetime larger than $\tau_{min} = 1.0$ are shown. Note that the streamlines have been integrated starting from the remaining points in the original noisy field not in a smoothed representation. The same holds for the classification of the critical points. Of course, it is not possible to recover the original topology of the noise-free data set but the overall behavior is much more apparent.

As a second example we present the application of the scale-space tracking to a real CFD data set—a simulation of the flow past a circular cylinder. There are 141 critical points that can be detected in this data set which account for the complex flow topology in the wake behind the cylinder. Figure 7.3 shows the critical points detected in the flow field. Only the lower half of the 180^3 data set is shown

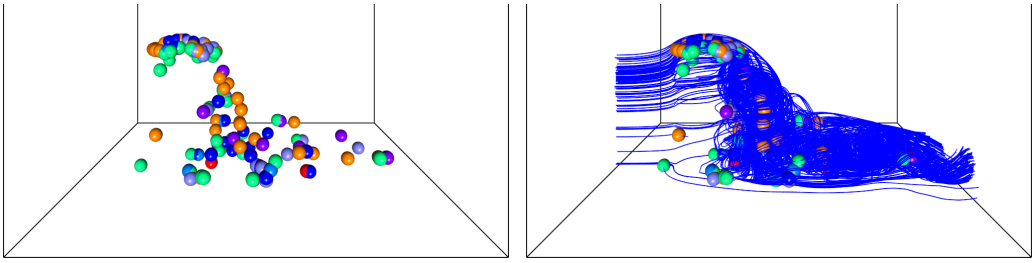


Figure 7.3: The flow behind a spherical cylinder. The same kind of color coding for the classification is used as for Figure 7.1. Left: Critical points on the finest scale, i.e. the original flow field. Right: Streamlines in the vicinity of the critical points.

in this image since there were no critical points detected in the upper part of the data set. The classification of the critical points is indicated by the same color scheme as for those shown in Figure 7.1. Computing the scale-space lifetime of the critical points for this data set in the interval $\tau = 0 \ldots 10$ took approximately 4:58 minutes using the same machine as for the first example. Since this data set is significantly larger than the artificial one used in the previous example and also contains far less critical points, most of the time (4:34 minutes) was spent for computing the scale-space flow field representation while for the relatively small number of critical points the scale-space tracking could be done in only 24 seconds. Seeding streamlines only around critical points that can be tracked for $\tau \geq 6.3$, as is shown in Figure 7.4, does significantly reduce the amount of streamlines that are displayed and accordingly the problems with visual clutter, but yet one can still clearly discern the overall behavior of the flow. Again, both tracking approaches produce comparable results. In comparison, using all detected critical points for streamline seeding, as is shown in Figure 7.3, leads to very densely packed streamlines that do not clearly convey the nature of the flow.

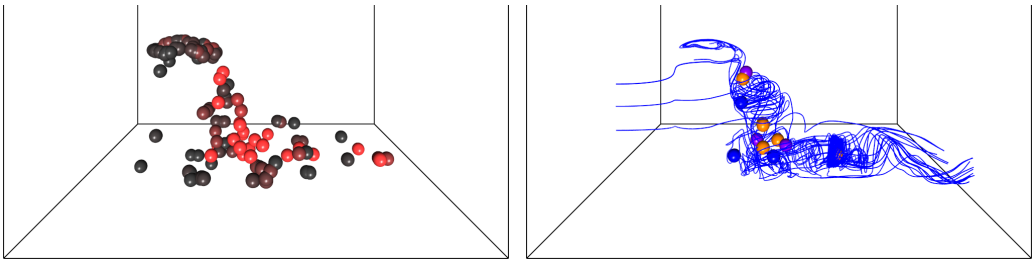


Figure 7.4: The flow behind a spherical cylinder. Left: Scale-space lifetime of the critical points. Right: Critical points were filtered by their lifetime ($\tau_{\text{min}} = 6.3$) and the remaining points have been used to seed streamlines in their vicinity.

7.6 Conclusion

In this chapter a first approach on tracking vector field singularities in scale space that uses explicit knowledge of the evolution of the field along the scale-axis has been investigated. The results of the experiments conducted with both artificial test data sets and real CFD data are promising. The complexity of the topology of noisy vector fields could be successfully reduced. The thus identified points of interest can be employed for seeding streamlines that capture the essential behavior of the flow or to place the point-based tensor glyphs described in the previous chapter.

Although the use of the linear scale space might not be the final answer for building a scale representation of a flow field, scale-space techniques are a promising way to deal with noisy and highly complex flow data sets. However, a rigorous analysis of the behavior of critical points of 3D vector fields under Gaussian blurring, similar to the work on scalar fields by Kuijper [115], has still to be done.

Chapter 8

Design Considerations for GPU-based Visualization Algorithms

Some of the approaches described in this thesis in particular those discussed in Chapter 5 have been already rendered obsolete by the fast paced development and the increased capabilities and resources of modern GPUs. However, there are still many traps that may trip the unwary. Many algorithms that can be nowadays written down in a straightforward manner in a high-level language like GLSL may, although they will work, not utilize the full computational potential of the hardware. Even more the simplified programming model may further lead developers towards an attitude of *the compiler will fix it* like we have seen it too often on the CPU side. Therefore, a profound understanding of the underlying hardware and the development of algorithms that fit a particular architecture is and will be also in the future necessary when designing GPU-based visualization algorithms. Thus, studying such techniques and how they exploit certain idiosyncrasies of graphics hardware can still prove beneficial for future developments.

The goal of this chapter is not to provide a general recipe for how to create efficient shader code nor is it meant to be a tutorial on how to optimize algorithms for a specific hardware. This is definitely beyond the scope of this thesis. Those recommendations would be very hardware architecture specific and would vary from hardware generation to hardware generation. What has been important on previous hardware architecture may prove irrelevant or even counterproductive on a new one or the model of another manufacturer. So hardware manufacturers know best and typically provide such information in the form of programming and optimization guidelines for each new generation of GPUs.

Instead, in the following sections a number of general strategies and considerations on the design of GPU-based visualization algorithms will be presented and

how these rules and guidelines have been employed in this work will be outlined. Furthermore, the problem of the lack of effective development tools will be addressed and a solution for interactive debugging of high-level shader code will be described briefly.

8.1 Exploiting Parallelism

Parallelism is widely considered the future of computing. Multiprocessor and multi-core processor architectures have already reached the PC desktop and are no longer a speciality of high-performance computing and large simulation systems. Since modern computer systems feature several independent processing units, writing efficient code means to exploit all of these units and to carefully balance the computational load between them.

Besides this coarse-grained task parallelism, which is the guiding characteristic of a modern shared-memory system and is also present in the stream processing model of parallel working graphics pipeline stages, the additional fine-grained data parallelism provided by a GPU on the per-vertex, per-fragment, or per-primitive level, poses additional advantages as well as difficulties. While current multi-core CPUs feature up to eight processor cores on a single chip, recent GPUs are build around a central shader core consisting of several hundred processing units. In comparison to commodity CPUs GPUs attain there computational power not from high clock rates but from the massively parallel deployment of relatively low-clocked processing units. However, while each CPU core can in general execute a different task, the data parallel computational model imposes more restrictions on the independent execution of tasks on a shader unit. Although the GPU is not strictly a SIMD array of processor elements but supports the more general *single program, multiple data* (SPMD) model, executing the same program at independent points on multiple data elements in parallel, the synchronisation points necessary for certain graphics pipeline features, such as computing the level of detail information for mipmapping, and the hardware overhead for a full program control unit on each parallel processing unit requires large batches of elements to be processed in lockstep mode for reasons of efficiency. Therefore, branch granularity is still a problem on GPUs and coherent branching, i.e. the same execution path being followed in the shader program by large blocks of data elements, is a prerequisite for fully exploiting the parallel processing capabilities of the hardware.

Thus, an algorithm that is considered to match onto the computational model of the GPU has to possess three important characteristics: a high degree of arithmetic computations per memory fetch, i.e. high arithmetic intensity, in order to keep the processing units busy, computational independence between processing

units, i.e. arithmetic occurs on each processing unit without being dependent on results of any other processing unit, and locally coherent branching characteristics.

Examples for the exploitation of parallel processing can be found in all of the preceding chapters and is a central theme of this thesis. The pixel-parallel ray casting discussed in Chapter 4 is one example. Here, the efficiency of certain optimization techniques, such as early ray termination or selective supersampling, are strongly dependent on the branch granularity required by the employed GPU. Like all rays are conceptually cast in parallel in GPU-based ray casting, the isosurface intersections with a million tetrahedra are computed in parallel on the fragment processing units in Chaper 5. The rendering of point-based glyphs discussed in Chapter 6, which exploits parallelism in vertex processing as well as for the pixel-parallel ray-surface intersection computation, is another example. And last, the detection of critical points, the computation of the scale-space representation, and the streamline integration necessary for tracking the vector field singularities in Chapter 7, are also good candidates for parallelization on the GPU. Currently this has been implemented on the CPU exploiting task parallelism on multiprocessor and multi-core architectures to speed up the computations.

8.2 Considering Computational Frequency

GPUs are designed with the primary goal of fast and efficient rendering of small to medium-sized textured polygonal models into a viewport that is displayed on a high-resolution display device. Consequently, the expected relative number of processed primitives and vertices is usually much smaller than the number of fragments that are generated. This has been reflected in the design of graphics processing units. The vertex processing units of former programmable GPUs were not as redundantly dimensioned as the fragment processing units, for example the NVIDIA NV40 graphics architecture provides sixteen fragment processors but only six vertex processing units. This disparity has been one of the primary reason for computing isosurface polygons on the fragment unit in Chapter 5. Consequently, considering the frequency of a computation, i.e. how frequently each value that is used in a shader needs to be computed, is a crucial aspect of GPU programming.

With the introduction of unified shader architectures in the recent graphics processor generations, the distinction between dedicated vertex and fragment processing units has been abandoned. Thus, unified architectures require additional care when balancing the workload between the stages of the pipeline. Since there is no explicit control by the programmer how shader units are scheduled for the different pipeline stages, i.e. how the shader units are shared between the frag-

ment, primitive, and vertex pipeline, it becomes even more complex to decide how to distribute the work between the programmable as well as the nonprogrammable stages of the pipeline. So, for example, it is no longer reasonable to consider vertex processing coming for free in a fragment processing bound case or vice versa. Nevertheless, in many cases it is economic to move computations to an earlier pipeline stage if they are the same for a large batch of computational elements. For example moving computations from the fragment to the vertex shader if their results are constant for a whole primitive or exploiting the interpolation capabilities of the rasterization unit for values linearly varying over a primitive can provide large savings on per-fragment processing costs. For example, this optimization strategy has been applied in the point splatting approach described in Chapter 6 by moving constant factors of the ray-surface intersection computation to the vertex shader. Furthermore it has been shown in this chapter that it might be beneficial to invest into elaborate primitive processing in order to reduce the fragment processing load.

8.3 Data Representation

Another important decision when designing a GPU-based algorithm is how to represent visualization data efficiently in GPU memory. This task is closely related to choosing the appropriate processing unit or pipeline stage and how to operate on the data. Input data can be stored in GPU memory in several different formats and internal representations. From the view of a graphics API these are textures, vertex attribute buffers, index buffers, and shader constant arrays.

Although in general all these GPU data structures represent arrays of opaque data elements, they differ in their typical access pattern. Vertex attribute buffers are conceptually 1D arrays that are typically accessed in linear order. Although it is also possible to achieve nonuniform access pattern by using index arrays, real random access using, for example, indices computed in a vertex shader program is not possible.

Textures are more flexible in several regards. First, they come as 1D, 2D, and 3D arrays and can be indexed accordingly. Furthermore, they provide true random access to their elements through dependent texture lookups using shader-computed texture coordinates that may also depend on values retrieved by previous texture fetches. This possibility allows to build complicated and hierarchical data structures, like skip-lists or trees, using indexed representations. The data encoding scheme for unstructured tetrahedral grids described in Chapter 5 is an example for such an indexed, hierarchical data structure. The most important property of textures, however, is their support for linear, bilinear, and trilinear interpolation. Leveraging the optimized texture interpolation hardware of the GPU

is at the core of many GPU-accelerated visualization techniques. A prime example is hardware-assisted direct volume rendering. As has been shown in Chapter 4 the performance of slice-based volume rendering as well as ray casting is dominated by the cost of data access and trilinear interpolation. Efficient trilinear interpolation is therefore the key to interactive direct volume rendering.

Textures can be further used as intermediate data buffers. That means their content can be updated by values computed in a fragment shader. The possibility to render directly into a texture allows efficient implementations of multi-pass algorithms, such as those presented in Chapter 4 and Chapter 6.

Another aspect that requires careful consideration is the choice of internal data formats. As a general rule it can be said that the lowest precision that is sufficient to represent the data should be used. In many cases 16-bit floating point or integer types are sufficient. This not only reduces the amount of memory for storing the data and saves on memory bandwidth when uploading it to the GPU but has also an impact on processing performance since accessing 8-bit or 16-bit data is still considerably faster than reading 32-bit floating point values. Thus, high-precision float data should be only used where absolutely necessary.

8.4 Avoiding Bottlenecks

Exploiting the full potential of graphics hardware is only possible if the available computational units can be fully utilized. However, high throughput and accordingly high performance can be only achieved, if a continuous data flow through the rendering pipeline is guaranteed. However, this is only possible if the interactions of the different GPU-pipeline stages are taken into account and congestions in the pipeline are avoided by carefully balancing the usage of its functional units and the bandwidth required for communicating the necessary data. Thus, two categories of bottlenecks can be identified: communication bottlenecks and computation bottlenecks.

As mentioned above, avoiding computation bottlenecks is primarily a matter of carefully pondering computational frequency. The question is which parts of an algorithm are best realized using the vertex shader, the geometry shader, the fragment shader, fixed function units, like rasterization and blending units, or the host system's CPU.

Communication bottlenecks can occur on multiple levels. When transferring data from main memory to the GPU, when accessing data in graphics memory, or when communicating data between pipeline stages.

First, the available graphics bus bandwidth has not increased at the same pace and to the same degree as has the computational power of GPUs or the bandwidth of both main memory and graphics memory. Currently the theoretical graphics

bus bandwidth is 4 GB/s for a graphics card connected via a 16-lane PCIe graphics interconnect while the theoretical peak performance of commodity GPUs has already reached the 1 TFlops range with a graphics memory bandwidth of roughly 100 GB/s.[I] So, raw memory read performance is already 25 times higher than what could be delivered to the graphics card and raw single-precision processing is already nearly three orders of magnitude higher than what could be theoretically delivered to the processing units when streaming directly from main memory. However, these are purely theoretical values, in an actual algorithm typically only a fraction of the theoretical peak performance is actually achieved and the number of arithmetical operations on a single data value is much higher. Nevertheless, the graphics bus is in many cases the most severe bottleneck for a GPU-based visualization algorithm. So the best solution is to minimize the number of data transfers and if possible keep all data in local graphics memory. For simplicity, most algorithms discussed in this thesis expect that all data can be stored in graphics memory. However, it will not always be possible to keep all data in GPU memory. If repeated transfers from main memory to graphics memory cannot be avoided, there are several strategies to keep the costs low. These are the use of preselection or level-of-detail techniques on the CPU to reduce the data size, the optimization of data transfer size, i.e. using the lowest precision that is sufficient for representing the data, the optimization of bus utilization by transferring large data chunks, and the use of asynchronous transfer capabilities of modern GPUs for uploading data. A data minimization strategy and asynchronous uploads using OpenGL vertex buffer objects are employed in Chapter 6 for interactive streaming of time-dependent particle data to the GPU. A grid partitioning strategy and CPU-sided preselection of cells for the isosurface extraction algorithm described in Chaper 5 has been examined in [147].

Second, graphics memory bandwidth and latency are the limiting factors for many algorithms. As has been shown in Chapter 4, GPU-based ray casting as well as slice-based volume rendering is mostly limited by texture sampling. Using pre-integration further worsens the situation. The additional dependent texture lookups and the more or less random sampling of the pre-integration table leads to texture cache thrashing, which negatively affects performance. In general, algorithms exhibiting a high arithmetic intensity are best suited for implementation on the GPU, since memory access latency can be hidden by ALU operations.

Last, communicating too much data between pipeline stages can be also a bottleneck. In Chapter 6 it has been observed that emitting too many vertex attributes in a geometry shader will limit the performance although the load on the fragment processor is reduced.

[I] NVIDIA GeForce GTX 280: 933 GFlops single-precision peak performance, 102 GB/s memory bandwidth.

8.5 Knowing Your Hardware

Besides best practice in programming, which includes the application of common design patterns and optimization strategies, such as the elimination of common subexpressions or loop unrolling, it is nevertheless important to know and to exploit the idiosyncrasies of the GPU architecture in order to leverage the full potential of the hardware. Although high-level languages, like GLSL or Cg, are convenient for writing portable code because they hide the specifics of the hardware, it is likely that a high-level compiler might produce less than optimal code. There are several reasons why this might be the case: First, a compiler cannot perform certain optimizations that might be in general unsafe but are valid in certain situations due to a priori knowledge about the data. Second, it may be beneficial to reformulate a solution in a way that better fits the available hardware resources or exploits certain characteristics or features of the hardware. An example for such a program transformation can be found in Chapter 4. Here, the evaluation of the termination criterion of the ray traversal loop is reformulated to reduce the number of necessary instructions by exploiting the automatic color saturation modifier supported by most GPUs, which provides a speedup of up to 20%.

While this optimization can be clearly measured in the number of shader instructions that could be saved, for other techniques it is not that obvious whether they work for a given hardware architecture or not. A well-known example of an optimization strategy, which has been common on older hardware but has been rendered obsolete by the increase in arithmetic processing performance of modern GPUs, is the method of replacing arithmetic instructions by texture lookups. While the precomputation of complex functions, including shading or even vector normalization, was a standard optimization on graphics hardware only few years ago on today's hardware the original benefit has completely switched to the opposite. On current hardware texture access latency is often the limiting factor. Furthermore, such an approach can lead to cache-thrashing with negative impact on the overall algorithm. On the other hand, precomputation can be also beneficial. As has been discussed in Chapter 4, using pre-integration the number of 3D sampling operations can be in many cases drastically reduced without negative impact on the volume rendering quality.

8.6 Development Tools

Last but not least, developing GPU-based algorithms is impossible without proper development tools. In general, the basic requirement for software development is nothing else as a text editor and a compiler and linker tool. However, in general, if the problem becomes more involved and accordingly the source code complexity

increases, the programming task becomes increasingly intricate and error prone and, thus, more sophisticated development tools are required to achieve correct and efficient code. No wonder, debuggers and profilers have become the two most important development tools besides the compiler.

However, in case of programmable graphics hardware, the development environment, in particular debugging and code analysis tools, did not keep up with the rapid advances of the hardware and software interface. As a result, developers typically spend quite a large amount of time locating and debugging programming errors. Thus, the availability of effective debugging means is essential for fast and efficient shader code development.

In comparison to a traditional debugger for sequential CPU programs, the special characteristics of graphics hardware pose additional challenges for a shader debugger. Most important, there is no direct access to the hardware, i.e. there is no specific low-level debugging interface. Second, it has to deal with the intrinsic parallelism of the graphics hardware that requires to deal with thousands or even millions of threads running conceptually in parallel.

In general, solutions for shader debugging can be divided into two basic approaches: software emulation and shader code instrumentation. The former uses a software implementation of the rendering pipeline in order to emulate the execution of shader programs according to the specification of the shading language. This allows for direct control of program execution and provides access to arbitrary data content. The main drawback of this approach is that debugging is not performed on the target hardware and results may therefore not correspond to actual hardware values. In contrast, shader code instrumentation will provide true hardware values, but access to the data is complicated since the hardware is in general only accessible through a high-level programming API. Since there is no direct access to the individual processing elements, the only possibility to get data back from the GPU is to read back the final result of a shader invocation, i.e. vertex attributes or pixel color. Accordingly, it is common practice in shader development to use printf-style visual debugging by manually rewriting fragment shaders to return the value of interest as its final output and to interpret the resulting color images. However, in the context of multi-pass rendering algorithms involving off-screen render targets or if debugging full floating-point precision data, the direct display of intermediate results is often not sufficient and a fair amount of code changes to the host program are required to permit readback of rendering results to main memory. This is even worse for vertex and geometry processing units, as they do not output directly displayable content and debugging those shaders may require additional changes in subsequent stages of the rendering pipeline. Furthermore, this manual *Edit&Continue* style of debugging is tedious and error prone.

An efficient shader debugging tool therefore must be able to automatically in-

strument shader code and host program in an application-transparent manner. In particular, it has to work without explicit code changes, i.e. it must not require modification and re-compilation of the host program. In the course of this thesis such a tool for debugging OpenGL Shading Language programs has been developed [202] and released to the public.[II] It provides a generic, minimal intrusive, and light-weight solution for debugging high-level shader programs directly on the target hardware. Furthermore, it operates completely application-transparent, i.e. it does not require any code changes to or re-compilation of the debugged target application. Including support for the full GLSL 1.2 specification for vertex and fragment shaders and the extensions for geometry shaders it completes the tool chain for OpenGL 2.1 shader development. Two examples of typical debug sessions are shown in Figure 8.1.

Instrumenting the host application is achieved using a dynamic library running in the process space of the debugged host application, providing full access to the process image and all OpenGL calls invoked by the target application without the need to recompile it or even having its source code available. The concept is based on interactively intercepting all OpenGL calls invoked by the application during its execution with full access to all function parameters, thus enabling among other things the retrieval of shader source code. Depending on the current state of the debugger this allows for either running the host program unaltered with only little performance overhead or stopping and stepping through the execution of the target program on a per OpenGL function call level. The later is used to identify a single draw call as target operation for shader debugging. In addition, the instrumenting library establishes a debugging environment, i.e. float buffer objects, in the graphical context of the host application that permits not only the retrieval of the requested debug results but also assures that subsequent program execution is not affected by the debugging process.

The actual debugger application can be divided into two components. First, the GLSL shader code instrumentation performs automatic code manipulation in order to inject additional debug statements. An OpenGL shading language parser is used to create an intermediate representation for a given shader that serves as basis for identifying the program execution order, variable scope determination, and the manipulation of shader source code in a syntactically and semantically correct manner. The compiler back end is a GLSL code generator that reconstructs valid shader programs from the intermediate language that can be used to replace the original shader code in the target's OpenGL state. Second, a graphical debugging interface, which serves the purpose of controlling the program execution of the host application, selecting the draw call of interest from the OpenGL command stream, specifying debug requests for shader variable data, and providing capable

[II] http://www.vis.uni-stuttgart.de/glsldevil

analysis and interaction methods for interactive visual debugging. More detail on the actual host application instrumentation and automatic shader instrumentation can be found in the original publication that describes the debug system [202].

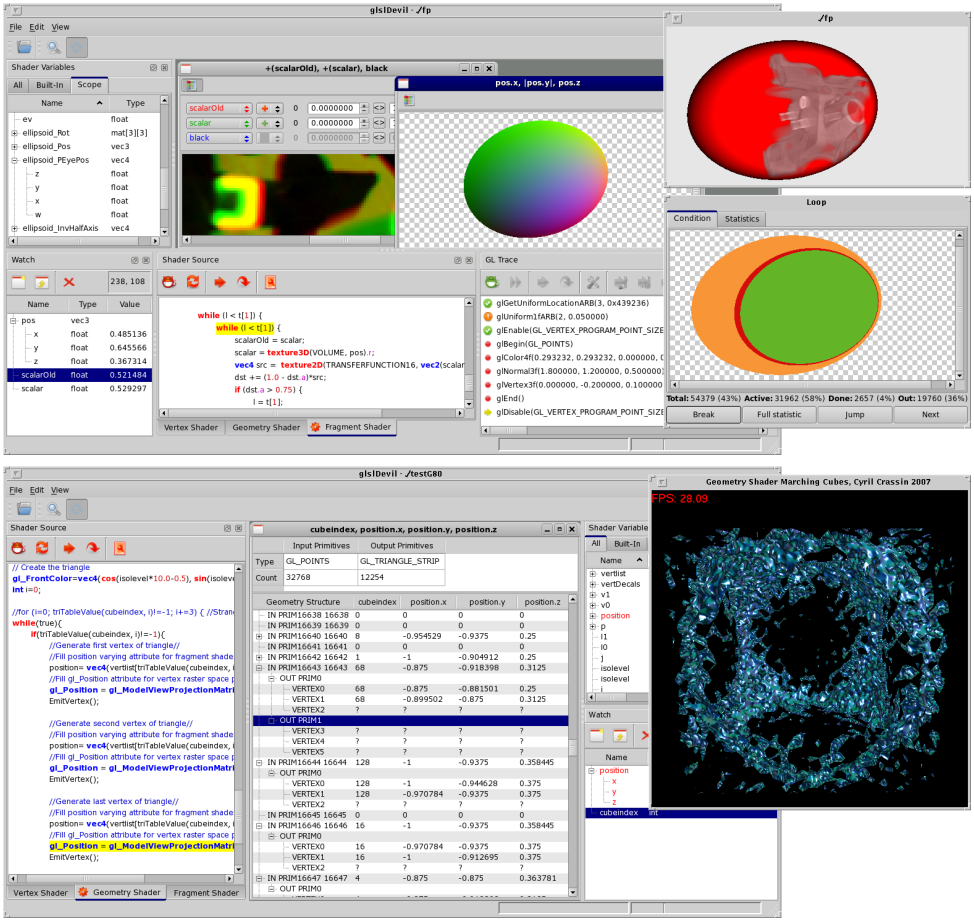


Figure 8.1: Examples of typical debugging session. In the upper image a fragment shader is debugged. From the original application's OpenGL stream a single draw call is selected and the active shader is debugged having source code, scope lists, variable data content, and graphical analysis tools at hand. Status information of the currently debugged loop statement supports the user in program flow decisions. The lower image shows the debugging of a geometry shader. Data inspection in this case is performed on a per input primitive level, showing for each input primitive the resulting output primitives and their corresponding vertices. Variable content at the current debug position is shown for each input primitive. Additionally, for already emitted vertices the variable values at the time of emission are shown. Question marks indicate output vertices that are not yet emitted.

Chapter 9

Contributions and Outlook

In this thesis several techniques for the visualization of three-dimensional data fields have been presented. The common tenor of these methods is the concept of exploiting modern programmable commodity graphics hardware for interactive visualization. In the following the contributions will be summarized and some directions for future research will be given.

9.1 Contributions

In the preceding chapters GPU-based visualization methods, ranging from global direct visualization techniques, over the extraction and visualization of derived geometric representations to feature-based abstract visualization methods, have been described. The three most common types of three-dimensional fields in academia and industry, i.e. scalar fields, vector fields, and symmetric tensor fields, have been targeted.

First, a single-pass GPU-based volume ray casting approach has been presented and its advantages over the standard slice-based volume rendering approach in terms of simplicity and flexibility have been discussed and have been highlighted by the demonstration of a number of standard and nonstandard volume rendering modalities, including pre-integrated direct volume rendering, shaded isosurfaces, and an extension to spectral refractive volume ray casting.

For evaluation of the single-pass ray casting approach a versatile, portable, and expandable volume visualization framework has been implemented that allows for fast and flexible prototyping of GPU-based visualization techniques and offers the opportunity for evaluation and comparison of different techniques or implementations. Using this framework, GPU-based ray casting has been evaluated in terms of image fidelity and rendering performance and has been compared against GPU-assisted slice-based volume rendering using viewport-aligned slices. It could be

shown that basic single-pass ray casting in terms of rendering speed performs equally well as slice-based volume rendering, but for a given sampling rate typically provides significantly higher image fidelity than slice-based rendering even when high-precision floating point buffers are used for blending. Furthermore, it has been shown that acceleration techniques, such as early ray termination, empty-space leaping, and adaptive sampling, can be realized with very little additional effort and that these optimizations, which are hard to implement in a slice-based volume renderer, can provide a significant performance benefit.

Second, it has been shown that creation of geometry is feasible on the GPU. The presented hardware-accelerated isosurface reconstruction approach is capable of extracting the polygonal isosurface geometry from arbitrary unstructured, nonconvex grids and storing it directly into graphics memory buffers. This opens new possibilities for generating polygonal isosurfaces from data available only in graphics memory without expensive readback to main memory or for postprocessing of the isosurface mesh on the GPU.

Third, a glyph-based visualization technique for symmetric, second-order tensor field data has been presented. The proposed point-based approach does not depend on tesselated geometry but reduces the necessary geometric data to the bare minimum, which drastically reduces the load on the graphics bus and allows the streaming of time-dependent data for interactive visualization of large numbers of individual glyphs.

Fourth, the widely used tensor ellipsoid paradigm has been extended for the visualization of indefinite tensor fields. A novel tensor glyph has been proposed that combines the advantages of the ellipsoid, i.e. finite extent, proportionality between eigenvalue magnitudes and glyph dimensions, and efficient point-based rendering, and the positive aspects of the general stress quadric, i.e. additional degrees of freedom for controlling the shape that allow to depict also indefinite tensors. Furthermore, the combination of the glyph rendering with the GPU-based ray casting technique provides additional context information and, thus, allows to reduce visual clutter by limiting the number of glyphs to those corresponding to tensors of certain important characteristics.

Fifth, a new physically-motivated measure for shear strain magnitude in flow fields has been derived. The root-mean-square of the shear strain magnitudes of all possible shear directions for a given point is used as a measure for the average shear stress acting in this point. Visualizing shear layers as isosurfaces of the thus obtained scalar field in combination with quadric tensor glyphs depicting the local properties of the shear strain tensor field allows to gain new insight into CFD simulation data.

Sixth, a novel method for extracting the global topological structure from a flow field has been proposed. This technique is specifically aimed for noisy vector fields with complicated small-scale topological features. By tracking the evo-

lution of vector singularities through scale-space it is possible to identify a set of critical points that dominate the qualitative global behavior of the flow. This information can in turn be used to steer other geometric or global visualization methods. Topology-guided streamline seeding or glyph placement are only two examples.

9.2 Outlook

In this thesis, solutions for the interactive visualization of medium-size to large data sets have been presented. However, the definition of large is always relative. What is considered large depends on the visualization problem as well as the current state of acquisition and simulation technology. A 512^3 16-bit volume data set can no longer be seriously considered large, while a 512^3 vector or tensor field given in single-precision floating point values can be still challenging on the average contemporary workstation or desktop PC. Furthermore, considering time-dependent data multiplies the size of data sets by orders of magnitudes. Dealing with really large data sets therefore requires sophisticated out-of-core processing to be feasible on a single system. Another solution is to move to the next level of parallelism, that is using distributed processing. Splitting the work among multiple identical processing units in a single system or across several compute nodes of a GPU cluster system combines the available memory and computational resources. Although in this work the problem of dealing with very large data sets that cannot be fit into the locally available graphics memory has not been directly addressed, all presented techniques can be easily generalized and implemented in a parallel fashion on a GPU cluster computer.

On the other hand, it can be expected that the future will bring substantial changes on the hardware and software side. Massive multi-core processor architectures and stream computing architectures, such as GPUs, are the keywords of current and future high-performance computing. Visualization, as a equally compute intensive and bandwidth hungry application, has obviously to follow the same path and some examples have been shown in the preceding chapters. However, there is much dynamics in the graphics hardware industry. New hardware architectures, like Intel's *Larrabee* design, are about to enter the market and present new software challenges that must be overcome to fully take advantage of their processing capabilities. Furthermore, new programming paradigms for GPUs that differ considerably from common graphics programming APIs, such as OpenGL or DirectX, have already emerged. General purpose programming APIs, such as CUDA or Brook+, provide more direct and efficient access to the processing units of the hardware and have already proven the feasibility of GPUs as an alternative or complement to multi-core CPUs for many problems in computer vision, simu-

lation, or visualization. So the use of the more general compute APIs to reduce the overhead of a graphics API will be the logical consequence for future GPU-based visualization algorithms.

Bibliography

[1] A. Abdul-Rahman and M. Chen. Spectral Volume Rendering Based on the Kubelka-Munk Theory. *Computer Graphics Forum*, 24(3):413–422, 2005.

[2] Advanced Micro Devices, Inc. AMD Stream Computing Webpage. Available at http://ati.amd.com/technology/streamcomputing/, August, 2008.

[3] Advanced Micro Devices, Inc. R600-Family Instruction Set Architecture. Available at http://developer.amd.com/documentation/guides/, August, 2008.

[4] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Towards a comprehensive volume visualization system. In *Proceedings of IEEE Visualization '92*, pages 13–20, 1992.

[5] C. Bajaj, P. Djeu, V. Siddavanahalli, and A. Thane. TexMol: Interactive Visual Exploration of Large Flexible Multi-Component Molecular Complexes. In *Proceedings of IEEE Visualization '04*, pages 243–250, 2004.

[6] P. J. Basser, J. Mattiello, and D. LeBihan. MR Diffusion Tensor Spectroscopy and Imaging. *Biophysical Journal*, 66(1):259–267, 1994.

[7] P. J. Basser and C. Pierpaoli. Microstructural and Physiological Features of Tissues Elucidated by Quantitative-Diffusion-Tensor MRI. *Journal of Magnetic Resonance, Series B*, 111(3):209–219, 1996.

[8] D. Bauer and R. Peikert. Vortex Tracking in Scale-Space. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '02*, pages 233–240, 2002.

[9] W. Baxter, J. Wendt, and M. C. Lin. IMPaSTo: A Realistic, Interactive Model for Paint. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 45–148, 2004.

[10] S. Bergner, T. Möller, and M. Drew. Spectral Volume Rendering. Technical Report SFU-CMPT-03/02-TR2002-03, School of Computing Science, Simon Fraser University, 2002.

[11] S. Bergner, T. Möller, M. S. Drew, and G. D. Finlayson. Interactive spectral volume rendering. In *Proceedings of IEEE Visualization '02*, pages 101–108, 2002.

[12] S. Bergner, T. Möller, M. Tory, and M. Drew. A Practical Approach to Spectral Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(2):207–216, 2005.

[13] S. Bergner, T. Möller, D. Weiskopf, and D. J. Muraki. A spectral analysis of function composition and its implications for sampling in direct volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1353–1360, 2006.

[14] F. F. Bernadon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. GPU-Based Tiled Ray Casting using Depth Peeling. *Journal of Graphics Tools*, 11(4):1–16, 2006.

[15] I. Bitter, N. Neophytou, K. Mueller, and A. E. Kaufman. Squeeze: Numerical-Precision-Optimized Volume Rendering. In *Proceedings of Graphics Hardware '04*, pages 25–34, 2004.

[16] J. F. Blinn. Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics (Proceedings of SIGGRAPH '77)*, 11(2):192–198, 1977.

[17] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer*, 17(3):185–197, 2001.

[18] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH '03)*, 22(3):917–924, 2003.

[19] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-Quality Surface Splatting on Today's GPUs. In *Proceedings of the Symposium on Point-Based Graphics '05*, pages 17–24, 2005.

[20] M. Botsch and L. Kobbelt. High-Quality Point-Based Rendering on Modern GPUs. In *Proceedings of Pacific Graphics '03*, pages 335–343, 2003.

[21] M. Botsch, M. Spernat, and L. Kobbelt. Phong Splatting. In *Proceedings of the Symposium on Point-Based Graphics '04*, pages 25–32, 2004.

[22] C. Boyd. The DirectX 11 Compute Shader. In *SIGGRAPH '08: ACM SIGGRAPH 2008 Classes, Beyond Programmable Shading: Fundamentals*, 2008.

[23] L. Buatois, G. Caumon, and B. Lévy. GPU Accelerated Isosurface Extraction on Tetrahedral Grids. In *Lecture Notes in Computer Science, Advances in Visual Computing (Proceedings of the 2nd International Symposium on Visual Computing)*, volume 4291/2006, pages 383–392. Springer Verlag, Berlin / Heidelberg, 2006.

[24] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '04)*, 23(3):777–786, 2004.

[25] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In *Proceedings of the IEEE Symposium on Volume Visualization '94*, pages 91–98, 1994.

[26] H. Carr, T. Möller, and J. Snoeyink. Artifacts Caused by Simplicial Subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, 2006.

[27] H. Chernoff. The Use of Faces to Represent Points in K-Dimensional Space Graphically. *Journal of the American Statistical Association*, 68(342):361–368, 1973.

[28] ClearSpeed Technology Plc. CLEARSPEED WHITEPAPER: CSX PROCESSOR ARCHITECTURE. Available at http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf, August, 2008.

[29] C. S. Co, B. Hamann, and K. I. Joy. Iso-Splatting: A Point-based Alternative to Isosurface Visualization. In *Proceedings of Pacific Graphics '03*, pages 325–334, 2003.

[30] D. Cohen and Z. Sheffer. Proximity clouds—an acceleration technique for 3d grid traversal. *The Visual Computer*, 11(1):27–38, 1994.

[31] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993.

[32] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, and D. H. Salesin. Computer-generated Watercolor. In *Proceedings of ACM SIGGRAPH '97*, pages 421–430, 1997.

[33] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proceedings of the ACM Workshop on Volume Visualization '92*, pages 91–98, 1992.

[34] W. de Leeuw and R. van Liere. Visualization of Global Flow Structures Using Multiple Levels of Topology. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '99*, pages 45–52, 1999.

[35] W. de Leeuw and J. van Wijk. A Probe for Local Flow Field Visualization. In *Proceedings of IEEE Visualization '93*, pages 39–45, 1993.

[36] T. Delmarcelle and L. Hesselink. Visualization of second order tensor fields and matrix data. In *Proceedings of IEEE Visualization '92*, pages 316–323, 1992.

[37] T. Delmarcelle and L. Hesselink. Visualizing second-order tensor fields with hyperstreamlines. *IEEE Computer Graphics and Applications*, 13(4):25–33, 1993.

[38] T. Delmarcelle and L. Hesselink. A unified framework for flow visualization. In R. S. Gallagher, editor, *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis*, pages 129–170. CRC Press, 1995.

[39] K. Devlin, A. Chalmers, A. Wilkie, and W. Purgathofer. Star: Tone reproduction and physically based spectral rendering. In *State of the Art Reports, Eurographics '02*, pages 101–123, 2002.

[40] R. R. Dickinson. A Unified approach to the design of visualization software for the analysis of field problems. In *Proceedings of SPIE, Three-Dimensional Visualization and Display Technologies*, volume 1083, pages 173–180, 1989.

[41] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE TRANSACTIONS on Information and Systems*, E74-D(1):214–224, 1991.

[42] J. Dorsey and P. Hanrahan. Modeling and Rendering of Metallic Patinas. pages 387–396, 1996.

[43] DRC Computer Corporation. DRC Reconfigurable Processor Units Product Information Webpage. http://www.drccomputer.com/drc/modules.html, August, 2008.

[44] M. J. Düurst. Additional Reference to "Marching Cubes". *Computer Graphics*, 22(2):72, 1988.

[45] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. HistoPyramids in Iso-Surface Extraction. Technical Report MPI-I-2007-4-006, Max-Planck-Institut für Informatik, Saarbrücken, 2007.

[46] D. Eberly. Perspective Projection of an Ellipsoid. Technical report, Geometric Tools, Inc., 1999. Available at http://www.geometrictools.com/Documentation/PerspectiveProjectionEllipsoid.pdf.

[47] D. S. Ebert, R. M. Rohrer, C. D. Shaw, P. Panda, J. M. Kukla, and D. A. Roberts. Procedural shape generation for multi-dimensional data visualization. *Computers & Graphics*, 24(3):375–384, 2000.

[48] D. S. Ebert and C. D. Shaw. Minimally Immersive Flow Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):343–350, 2001.

[49] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, 2006.

[50] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings of Graphics Hardware '01*, pages 9–16, 2001.

[51] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman, Amsterdam, 2003.

[52] S. E. Fienberg. Graphical Methods in Statistics. *The American Statistician*, 33(4):165–178, 1979.

[53] L. Florack and A. Kuijper. The Topological Structure of Scale-Space Images. *Journal of Mathematical Imaging and Vision*, 12(1):65–79, 2000.

[54] B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988.

[55] D. Frederick and T. S. Chang. *Continuum mechanics*. Allyn and Bacon, Inc., Boston, 1965.

[56] M. P. Garrity. Raytracing irregular volume data. In *Proceedings of the ACM Workshop on Volume Visualization '90*, pages 35–40, 1990.

[57] C. Garth, X. Tricoche, and G. Scheuermann. Tracking of Vector Field Singularities in Unstructured 3D Time-Dependent Datasets. In *Proceedings of IEEE Visualization '04*, pages 329–336, 2004.

[58] A. S. Glassner. How to Derive a Spectrum from an RGB Triplet. *IEEE Computer Graphics and Applications*, 9(4):95–99, 1989.

[59] A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, 1995.

[60] A. Globus, C. Levit, and T. Lasinski. A Tool for Visualizing the Topology of Three-Dimensional Vector Fields. In *Proceedings of IEEE Visualization '91*, pages 33–40, 1991.

[61] M. Gschwind. The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[62] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Springer Verlag, 1986.

[63] B. Gudmundsson and M. Randén. Incremental Generation of Projections of CT-Volumes. In *Proceedings of The First Conference on Visualization in Biomedical Computing*, pages 27–34, 1990.

[64] S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '03*, pages 245–252, 2003.

[65] S. Guthe, S. Gumhold, and W. Strasser. Interactive visualization of volumetric vector fields using texture based particles. *Journal of WSCG*, 10(3):33–41, 2002.

[66] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization '02*, pages 53–60, 2002.

[67] S. Guy and C. Soler. Graphics gems revisited: Fast and physically-based rendering of gemstones. *ACM Transactions on Graphics*, 23(3):231–238, 2004.

[68] C. Haase and G. Meyer. Modeling Pigmented Materials for Realistic Image Synthesis. *ACM Transactions on Graphics*, 11(4):305–335, 1992.

[69] R. B. Haber. Visualization Techniques for Engineering Mechanics. *Computing systems in engineering*, 1(1):37–50, 1990.

[70] R. B. Haber and D. A. McNabb. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In G. M. Nielson and B. Shriver, editors, *Visualization in Scientific Computing*, pages 74–93. IEEE Computer Scociety Press, 1990.

[71] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. H. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[72] R. Haimes and D. Kenwright. On the velocity gradient tensor and fluid feature extraction. *Proceedings of the 14th Computational Fluid Dynamics Conference, Norfolk, VA*, pages 315–324, 1999.

[73] R. A. Hall and D. P. Greenburg. A Testbed for Realistic Image Synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, 1983.

[74] E. Halley. An Historical Account of the Trade Winds, and Monsoons, Observable in the Seas between and Near the Tropicks, with an Attempt to Assign the Phisical Cause of the Said Winds. *Philosophical Transactions*, 16(183):153–168, 1686.

[75] A. Halm, L. Offen, and D. W. Fellner. BioBrowser: A Framework for Fast Protein Visualization. In *Proceedings of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EUROVIS) '05*, pages 287–294, 2005.

[76] Y. Hashash, J. Yao, and D. Wotring. Glyph and hyperstreamline representation of stress and strain tensors and material constitutive response. *International Journal for Numerical and Analytical Methods in Geomechanics*, 27(7):603–626, 2003.

[77] J. L. Helman and L. Hesselink. Representation and Display of Vector Field Topology in Fluid Flow Data Sets. *IEEE Computer*, 22(8):27–36, 1989.

[78] J. L. Helman and L. Hesselink. Visualizing Vector Field Topology in Fluid Flows. *IEEE Computer Graphics and Applications*, 11(3):36–46, 1991.

[79] L. Hesselink and T. Delmarcelle. Visualization of Vector and Tensor Datasets. In L. Rosenblum et al., editor, *Scientific Visualization: Advances and Challenges*, pages 419–433. Academic Press, 1994.

[80] L. Hesselink, Y. Levy, and Y. Lavin. The Topology of Symmetric, Second-Order 3D Tensor Fields. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):1–11, 1997.

[81] M. Hlawitschka, G. Scheuermann, and B. Hamann. Interactive Glyph Placement for Tensor Fields. In *Lecture Notes in Computer Science, Advances in Visual Computing (Proceedings of the 3rd International Symposium on Visual Computing)*, volume 4841/2007, pages 331–340. Springer Verlag, Berlin / Heidelberg, 2007.

[82] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of ACM SIGGRAPH '99*, pages 277–286, 1999.

[83] W. Hong, F. Qiu, and A. E. Kaufman. GPU-based Object-Order Ray-Casting for Large Datasets. In *Proceedings of the Eurographics / IEEE VGTC Workshop on Volume Graphics (VG) '05*, pages 177–185, 2005.

[84] M. Hopf and T. Ertl. Hardware-Based Wavelet Transformations. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '99*, pages 317–328, 1999.

[85] M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '00*, pages 337–345, 2000.

[86] T. Iijima. Basic theory on normalization of a pattern (in case of typical one-dimensional pattern). *Bulletin of Electrical Laboratory*, 26:368–388, 1962.

[87] T. J. Jankun-Kelly and K. Mehta. Superellipsoid-based, Real Symmetric Traceless Tensor Glyphs Motivated by Nematic Liquid Crystal Alignment Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1197–1204, 2006.

[88] B. Jeremić, G. Scheuermann, J. Frey, Z. Yang, B. Hamann, K. I. Joy, and H. Hagen. Tensor Visualization in Computational Geomechanics. *International Journal for Numerical and Analytical Methods in Geomechanics*, 26:925–944, 2002.

[89] G. Johansson and H. Carr. Accelerating Marching Cubes with Graphics Hardware. In *CASCON '06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, page 39, 2006.

[90] C. Johnson. Top Scientific Visualization Research Problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, 2004.

[91] C. R. Johnson, R. Moorhead, T. Munzner, H. Pfister, P. Rheingans, and T. S. Yoo, editors. *NIH-NSF Visualization Research Challenges Report*. IEEE Press, Los Alamitos, CA, USA, 1st edition, 2006.

[92] G. M. Johnson and M. D. Fairchild. Full-Spectral Color Calculations in Realistic Image Synthesis. *IEEE Computer Graphics and Applications*, 19(4):47–53, 1999.

[93] J. T. Kajiya and B. P. von Herzen. Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics (Proceedings of SIGGRAPH '84)*, 18(3):165–174, 1984.

[94] K. Kim, C. M. Wittenbrink, and A. Pang. Extended Specifications and Test Data Sets for Data Level Comparisons of Direct Volume Rendering Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):299–317, 2001.

[95] G. Kindlmann. Superquadric Tensor Glyphs. In *Proceedings of the Joint Eurographics IEEE/TVCG Symposium on Visualization (VisSym) '04*, pages 147–154, 2004.

[96] G. Kindlmann, D. Weinstein, and D. Hart. Strategies for Direct Volume Rendering of Diffusion Tensor Fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):124–138, 2000.

[97] G. Kindlmann and C.-F. Westin. Diffusion Tensor Visualization with Glyph Packing. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1329–1336, 2006.

[98] P. Kipfer and R. Westermann. GPU Construction and Transparent Rendering of Iso-Surfaces. In *Proceedings of the Workshop on Vision, Modeling and Visualization '05*, pages 241–248, 2005.

[99] T. Klein, M. Eissele, D. Weiskopf, and T. Ertl. Simulation, Modelling and Rendering of Incompressible Fluids in Real Time. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '03*, pages 365–373, 2003.

[100] T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '04*, pages 387–394, 2004.

[101] T. Klein and T. Ertl. Scale-Space Tracking of Critical Points in 3D Vector Fields. In H. Hauser, H. Hagen, and H. Theisel, editors, *Mathematics and Visualization, Topology-Based Methods in Visualization (Proceedings of TopoInVis '05)*, pages 35–49. Springer, Berlin Heidelberg, 2007.

[102] 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.

[103] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware. In *Proceedings of IEEE Visualization '05*, pages 223–230, 2005.

[104] B. Kleiner and J. A. Hartigan. Representing Points in Many Dimensions by Trees and Castles. *Journal of the American Statistical Association*, 76(374):260–269, 1981.

[105] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[106] J. J. Koenderink. The Structure of Images. *Biological Cybernetics*, 50:363–370, 1984.

[107] M. Kraus. *Direct Volume Visualization of Geometrically Unpleasant Meshes*. PhD thesis, Universität Stuttgart, 2003.

[108] M. Kraus, W. Qiao, and D. S. Ebert. Projecting Tetrahedra without Rendering Artifacts. In *Proceedings IEEE Visualization '04*, pages 27–34, 2004.

[109] M. Kraus and M. Strengert. Pyramid Filters Based on Bilinear Interpolation. In *Proceedings of the 2nd International Conference on Computer Graphics Theory and Applications*, volume GM/R, pages 21–28, 2007.

[110] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03*, pages 287–292, 2003.

[111] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '03)*, 22(3):908–916, 2003.

[112] P. Kubelka. New contributions to the optics of intensely light-scattering materials, Part I. *Journal of the Optical Society of America*, 38:448–457, 1948.

[113] P. Kubelka. New contributions to the optics of intensely light-scattering materials, Part II. Nonhomogeneous layers. *Journal of the Optical Society of America*, 44:330–355, 1954.

[114] P. Kubelka and F. Munk. Ein Beitrag zur Optik der Farbanstriche. *Zeitschrift für Technische Physik*, 12:593–601, 1931.

[115] A. Kuijper. *The Deep Structure of Gaussian Scale Space Images*. PhD thesis, Utrecht University, 2002.

[116] A. Kuijper and L. Florack. Calculations on Critical Points under Gaussian Blurring. In *Lecture Notes in Computer Science, Scale-Space Theories in Computer Vision (Proceedings of the 2nd International Conference on Scale-Space Theories in Computer Vision '99)*, volume 1682/1999, pages 318–329. Springer, Berlin / Heidelberg, 1999.

[117] S. Lakare and A. Kaufman. Light Weight Space Leaping Using Ray Coherence. In *Proceedings of IEEE Visualization '04*, pages 19–26, 2004.

[118] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution Techniques for interactive Texture-based Volume Visualization. In *Proceedings of IEEE Visualization '99*, pages 355–361, 1999.

[119] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing '01 (CDROM)*, page 55, 2001.

[120] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[121] C. Lepage, T. Leweke, and A. Verga. Spiral shear layers: Roll-up and incipient instability. *Physics of Fluids*, 17(3):031705, 2005.

[122] M. Levoy. Volume rendering: Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[123] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[124] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *Proceedings of ACM SIGGRAPH '00*, pages 131–144, 2000.

[125] M. Levoy and T. Whitted. The use of points as display primitive. Technical Report TR 85–022, University of North Carolina at Chapel Hill, 1985.

[126] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares. Efficient Implementation of Marching Cubes Cases with Topological Guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[127] S. Li and K. Mueller. Accelerated, high-quality refraction computations for volume graphics. In *Proceedings of Eurographics/IEEE VGTC Workshop on Volume Graphics '05*, pages 73–229, 2005.

[128] W. Li, K. Mueller, and A. Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *Proceedings of IEEE Visualization '03*, pages 317–324, 2003.

[129] T. Lindeberg. *Scale-Space Theory in Computer Vision*. Kluwer Academic Publishers, 1994.

[130] T. Lindeberg. Edge Detection and Ridge Detection with Automatic Scale Selection. *International Journal of Computer Vision*, 30(2):117–156, 1998.

[131] Y. Livnat, H.-W. Shen, and C. R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[132] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM SIGGRAPH Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):163–169, 1987.

[133] A. Love. *A Treatise on the Mathematical Theory of Elasticity*. Cambridge University Press, Cambridge, 1892.

[134] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, 2003.

[135] E. B. Lum, B. Wilson, and K.-L. Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proceedings of the Joint Eurographics IEEE/TVCG Symposium on Visualization (VisSym) '04*, pages 25–34, 2004.

[136] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering*, pages 15–22, 1993.

[137] R. Mace. *OpenGL ARB Superbuffers*. ATI Research, Inc., 2004. Available at http://www.ati.com/developer/gdc/SuperBuffers.pdf.

[138] S. Mann and A. Rockwood. Computing Singularities of 3D Vector Fields with Geometric Algebra. In *Proceedings of IEEE Visualization '02*, pages 283–290, 2002.

[139] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH '03)*, 22(3):896–907, 2003.

[140] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[141] N. Max. Hierarchical molecular modelling with ellipsoids. *Journal of Molecular Graphics*, 23(3):233–238, 2004.

[142] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *ACM SIGGRAPH Computer Graphics (Proceedings of the Workshop on Volume Visualization '90)*, 24(5):27–33, 1990.

[143] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proceedings of Graphics Hardware '02*, pages 57–68, 2002.

[144] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Computing*, 33(10-11):648–662, 2007.

[145] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Proceedings of Graphics Hardware '02*, pages 137–146, 2002.

[146] D. Meyer. *Direkte numerische Simulation nichtlinearer Transitionsmechanismen in der Strömungsgrenzschicht einer ebenen Platte*. PhD thesis, Universität Stuttgart, 2003.

[147] D. Mohr. Isoflächenrekonstruktion für unstrukturierte Gitter mit Hilfe von Graphikhardware. Diploma Thesis, Institut für Visualisierung und interaktive Systeme, Universität Stuttgart, 2005.

[148] J. Moore, S. Schorn, and J. Moore. Methods of classical mechanics applied to turbulence stresses in a tip leakage vortex. *Journal of Turbomachinery*, 118(4):622–629, 1996.

[149] C. Müller, M. Strengert, and T. Ertl. Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) '06*, pages 59–66, 2006.

[150] A. Munshi. OpenCL. In *SIGGRAPH '08: ACM SIGGRAPH 2008 Classes, Beyond Programmable Shading: Fundamentals*, 2008.

[151] H. Noordmans, H. van der Voort, and A. Smeulders. Spectral Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):196–207, 2000.

[152] NVIDIA Corporation. CUDA Programming Guide. Available at http://www.nvidia.com/object/cuda_develop.html, August, 2008.

[153] NVIDIA Corporation. NVIDIA SDK 8.0. Available at http://developer.nvidia.com/object/sdk_home.html, August, 2008.

[154] NVIDIA Corporation. PTX: Parallel Thread Execution, ISA Version 1.2. Available at http://www.nvidia.com/object/cuda_develop.html, August, 2008.

[155] NVIDIA Corporation. Technical Brief - The GeForce 6 Series of GPUs High Performance and Quality for Complex Image Effects. Available at http://www.nvidia.com/object/ feature_HPeffects.html, August, 2008.

[156] J. A. Osborne. Demagnetization Factors of the General Ellipsoid. *Physical Review*, 67(11):351–357, 1945.

[157] V. Pascucci. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Proceedings of the Joint Eurographics IEEE/TVCG Symposium on Visualization (VisSym) '04*, pages 293–300, 2004.

[158] M. S. Peercy. Linear Color Representations for Full Spectral Rendering. pages 191–198, 1993.

[159] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH '00*, pages 335–342, 2000.

[160] C. Pierpaoli and P. J. Basser. Toward a quantitative assessment of diffusion anisotropy. *Magnetic Resonance in Medicine*, 36(6):893–906, 1996.

[161] C. Pierpaoli, I. linfante, J. Mattiello, G. D. Chiro, D. L. Bihan, and P. J. Basser. Diffusion tensor imaging of brain white matter anisotropy. In *Proceedings of the International Society for Magnetic Resonance in Medicine, 2nd Meeting*, page 1038, 1994.

[162] F. L. Ponta. Effect of shear-layer thickness on the Strouhal Reynolds number relationship for bluff-body wakes. *Journal of Fluids and Structures*, 22:1133–1138, 2006.

[163] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650, 2007.

[164] F. H. Post, W. C. deLeeuw, I. A. Sadarjoen, F. Reinders, and T. van Walsum. Global, geometric, and feature-based techniques for vector field visualization. *Future Generation Computer Systems*, 15(1):87–98, 1999.

[165] F. H. Post, F. J. Post, T. V. Walsum, and D. Silver. Iconic Techniques for Feature Visualization. In *Proceedings of IEEE Visualization '95*, pages 288–295, 1995.

[166] 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.

[167] C. Poupon, J.-F. Mangin, V. Frouin, J. Régis, F. Poupon, M. Pachot-Clouard, D. L. Bihan, and I. Bloch. Regularization of MR Diffusion Tensor Maps for Tracking Brain White Matter Bundles. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 489–498, 1998.

[168] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime Isosurface Extraction with Graphics Hardware. In *Eurographics 2004, Short Presentations and Interactive Demos*, 2004.

[169] G. Reina, K. Bidmon, F. Enders, P. Hastreiter, and T. Ertl. GPU-Based Hyperstreamlines for Diffusion Tensor Imaging. In *Proceedings of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EUROVIS) '06*, pages 35–42, 2006.

[170] G. Reina and T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *Proceedings of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EUROVIS) '05*, pages 177–182, 2005.

[171] G. Reina, T. Klein, and T. Ertl. Visualization of Attributed 3D Point Datasets. In M. Gross and H. Pfister, editors, *Point-Based Graphics*, pages 420–435. Morgan Kaufmann Publishers, Burlington, MA, 2007.

[172] 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 *Proceedings of Graphics Hardware '00*, pages 109–118, 2000.

[173] D. Rodgman and M. Chen. Refraction in Discrete Raytracing. In *Proceedings of the Eurographics/IEEE TCVG Workshop on Volume Graphics '01*, pages 3–17, 2001.

[174] D. Rodgman and M. Chen. Refraction in volume graphics. *Graphical Models*, 68(5):432–450, 2006.

[175] R. J. Rost. *OpenGL® Shading Language (2nd Edition)*. Addison-Wesley Longman, Amsterdam, 2006.

[176] S. Röttger and T. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proceedings of the IEEE Symposium on Volume Visualization and Graphics '02*, pages 23–28, 2002.

[177] S. Röttger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '03*, pages 231–238, 2003.

[178] S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. In *Proceedings of IEEE Visualization '00*, pages 109–116, 2000.

[179] D. Rudolf, D. Mould, and E. Neufeld. Simulating Wax Crayons. In *Proceedings of Pacific Graphics*, pages 163–172, 2003.

[180] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH '00*, pages 343–352, 2000.

[181] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *ACM SIGGRAPH Computer Graphics*, 22(4):51–58, 1988.

[182] W. Sander and B. Weigand. Direct numerical simulation and analysis of instability enhancing parameters in liquid sheets at moderate Reynolds numbers. *Physics of Fluids*, 20(5):053301, 2008.

[183] Y. Sato, N. Shiraga, S. Nakajima, S. Tamura, and R. Kikinis. Local Maximum Intensity Projection (LMIP): A New Rendering Method for Vascular Visualization. *Journal of Computer Assisted Tomography*, 22(6):912–917, 1998.

[184] H. Schade. *Tensoranalysis*. de Gruyter, Berlin, 1997.

[185] G. Scheuermann, H. Hagen, H. Krüger, M. Menzel, and A. Rockwood. Visualization of Higher Order Singularities in Vector Fields. In *Proceedings of IEEE Visualization '97*, pages 67–74, 1997.

[186] G. Scheuermann, H. Krüger, M. Menzel, and A. P. Rockwood. Visualizing Nonlinear Vector Field Topology. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):109–116, 1998.

[187] G. Scheuermann, X. Tricoche, and H. Hagen. C1-Interpolation for Vector Field Topology Visualization. In *Proceedings of IEEE Visualization '99*, pages 271–533, 1999.

[188] W. J. Schroeder and K. M. Martin. Overview of Visualization. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 3–35. Academic Press, Orlando, FL, 2004.

[189] A. Schuster. Radiation Through a Foggy Atmosphere. *Journal of Astrophysics*, 21:1–22, 1905.

[190] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[191] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 23rd ACM National Conference*, pages 517–524, 1968.

[192] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM SIGGRAPH Computer Graphics (Proceedings of the Workshop on Volume Visualization '90)*, 24(5):63–70, 1990.

[193] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In M. Pharr, editor, *GPU Gems 2*, pages 313–329. Addison Wesley, 2005.

[194] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-Based Ray-Casting of Quadratic Surfaces. In *Proceedings of the Eurographics Symposium on Point Based Graphics '06*, pages 59–65, 2006.

[195] D. Silver, N. Zabusky, V. Fernandez, R. Samtaney, and M. Gao. Ellipsoidal Quantification of Evolving Phenomena. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 573–588. Springer-Verlag, Tokyo, 1991.

[196] M. Sramek and A. Kaufman. Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):236–252, 2000.

[197] S. Stegmaier. *Acceleration Techniques for Numerical Flow Visualization*. PhD thesis, Universität Stuttgart, 2006.

[198] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the Eurographics / IEEE VGTC Workshop on Volume Graphics (VG) '05*, pages 187–195, 2005.

[199] C. Stoll, S. Gumhold, and H.-P. Seidel. Visualization with stylized line primitives. In *Proceedings of IEEE Visualization '05*, pages 695–702, 2005.

[200] M. Strengert. Personal communication, 2007.

[201] M. Strengert, T. Klein, R. P. Botchen, S. Stegmaier, M. Chen, and T. Ertl. Spectral Volume Rendering using GPU-based Raycasting. *The Visual Computer*, 22(8):550–561, 2006.

[202] M. Strengert, T. Klein, and T. Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In *Proceedings of Graphics Hardware '07*, pages 81–88, 2007.

[203] Y. Sun, F. Fracchia, M. Drew, and T. Calvert. A spectrally based framework for realistic image synthesis. *The Visual Computer*, 17(7):429–444, 2001.

[204] M. Tabor. *Chaos and Integrability in Nonlinear Dynamics: An Introduction*. Wiley-Interscience, 1989.

[205] H. Theisel and H.-P. Seidel. Feature flow fields. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '03*, pages 141–148, 2003.

[206] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Saddle Connectors - An Approach to Visualizing the Topological Skeleton of Complex 3D Vector Fields. In *Proceedings of IEEE Visualization '03*, pages 225–232, 2003.

[207] M. Tory and T. Möller. A Model-Based Visualization Taxonomy. Technical Report CMPT-TR2002-06, Computing Science Department, Simon Fraser University, 2002.

[208] X. Tricoche, G. Scheuermann, and H. Hagen. Continuous Topology Simplification of Planar Vector Fields. In *Proceedings of IEEE Visualization '01*, pages 159–166, 2001.

[209] X. Tricoche, G. Scheuermann, and H. Hagen. Topology-Based Visualization of Time-Dependent 2D Vector Fields. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '01*, pages 117–126, 2001.

[210] X. Tricoche, T. Wischgoll, G. Scheuermann, and H. Hagen. Topology Tracking for the Visualization of Time-Dependent Two-Dimensional Flows. *Computer & Graphics*, 26(2):249–257, 2002.

[211] M. Üffinger, T. Klein, M. Strengert, and T. Ertl. GPU-Based Streamlines for Surface-Guided 3D Flow Visualization. In *Proceedings of the Workshop on Vision, Modelling, and Visualization '08*, 2008. to appear.

[212] V. Verma, D. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *Proceedings of IEEE Visualization '00*, pages 163–170, 2000.

[213] M. Wan, A. Sadiq, and A. Kaufman. Fast and Reliable Space Leaping for Interactive Volume Rendering. In *Proceedings of IEEE Visualization '02*, pages 195–202, 2002.

[214] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Proceedings of ACM SIGGRAPH '01*, pages 361–370, 2001.

[215] G. Ward and E. Eydelberg-Vileshin. Picture perfect rgb rendering using spectral prefiltering and sharp color primaries. In *Proceedings of the Eurographics Workshop on Rendering '02*, pages 117–124, 2002.

[216] M. Weiler, T. Klein, and T. Ertl. Direct volume rendering in OpenSG. *Computers and Graphics*, 28(1):93 – 98, 2004.

[217] M. Weiler, M. Kraus, and T. Ertl. Hardware-Based View-Independent Cell Projection. In *Proceedings of the IEEE Symposium on Volume Visualization and Graphics '02*, pages 13–22, 2002.

[218] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization '03*, pages 333–340, 2003.

[219] T. Weinkauf, H. Theisel, H.-C. Hege, and H.-P. Seidel. Boundary Switch Connectors for Topological Visualization of Complex 3D Vector Fields. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym) '04)*, pages 183–192, 2004.

[220] E. W. Weisstein. Hexagonal Close Packing. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/HexagonalClosePacking.html, 2008.

[221] R. Westermann and T. Ertl. Efficiently using Graphics Hardware in Volume Rendering Applications. In *Proceedings of ACM SIGGRAPH '98*, pages 169–177, 1998.

[222] C.-F. Westin, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and R. Kikinis. Processing and Visualization of Diffusion Tensor MRI. *Medical Image Analysis*, 6(2):93–108, 2002.

[223] C.-F. Westin, S. Peled, H. Gudbjartsson, R. Kikinis, and F. A. Jolesz. Geometrical Diffusion Measures for MRI from Tensor Basis Analysis. In *Proceedings of the ISMRM Scientific Meeting and Exhibition '97*, 1997.

[224] L. Westover. Footprint Evaluation for Volume Rendering. *ACM SIGGRAPH Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):367–376, 1990.

[225] T. Weyrich, S. Heinzle, T. Aila, D. B. Fasnacht, S. Oetiker, M. Botsch, C. Flaig, S. Mall, K. Rohrer, N. Felber, H. Kaeslin, and M. Gross. A Hardware Architecture for Surface Splatting. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH '07)*, volume 26, page 90, 2007.

[226] M. R. Wiegell, H. B. W. Larsson, and V. J. Wedeen. Fiber Crossing in Human Brain Depicted with Diffusion Tensor MR Imaging. *Radiology*, 217(3):897–903, 2000.

[227] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM SIGGRAPH Computer Graphics (Proceedings of the Workshop on Volume Visualization '90)*, 24(5):57–62, 1990.

[228] P. L. Williams and N. Max. A Volume Density Optical Model. In *Proceedings of the ACM Workshop on Volume Visualization '92*, pages 61–68, 1992.

[229] P. L. Williams and S. P. Uselton. Foundations for Measuring Volume Rendering Quality. Technical Report NAS-96-021, NASA Ames Research Center, 1996.

[230] O. Wilson, A. V. Gelder, and J. Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

[231] A. P. Witkin. Scale-Space Filtering. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 1019–1022, 1983.

[232] H. Wong, H. Qu, U. Wong, Z. Tang, and K. Mueller. A Perceptual Framework for Comparisons of Direct Volume Rendered Images. In *Proceedings of the IEEE Pacific-Rim Symposium on Image and Video Technology (PSIVT) '06*, pages 1314–1323, 2006.

[233] B. Wünsche. The Visualization and Measurement of Left Ventricular Deformation. In *Proceedings of the First Asia-Pacific Bioinformatics Conference (APBC) '03*, pages 119–128, 2003.

[234] G. Wyszecki and W. S. Stiles. *Color Science Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience Publication, 1982.

[235] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for *soft* objects. *The Visual Computer*, 2(4):227–234, 1986.

[236] R. Yagel and Z. Shi. Accelerating Volume Animation by Space-Leaping. In *Proceedings of IEEE Visualization '93*, pages 62–69, 1993.

[237] S. Yamakawa and K. Shimada. High Quality Anisotropic Tetrahedral Mesh Generation Via Ellipsoidal Bubble Packing. In *Proceedings of the 9th International Meshing Roundtable*, pages 263–274, 2000.

[238] L. Yang and B. Kruse. Revised Kubelka–Munk theory. I. Theory and application. *Journal of the Optical Society of America A*, 21(10):1933–1941, 2004.

[239] H. Yee. A perceptual metric for production testing. *Journal of Graphics Tools*, 9(4):33–40, 2004.

[240] I. Yoon, J. Demers, T. Kim, and U. Neumann. Accelerating Volume Visualization by Exploiting Temporal Coherence. In *Proceedings of IEEE Visualization '97, LBHT*, pages 21–24, 1997.

[241] K. Zuiderveld, A. Koning, and M. Viergever. Acceleration of Ray-casting using 3D Distance Transforms. In *Proceedings of Visualization in Biomedical Computing '92*, pages 324–335, 1992.

[242] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA Volume Splatting. In *Proceedings of IEEE Visualization '01*, pages 29–36, 2001.

[243] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface Splatting. In *Proceedings of ACM SIGGRAPH '01*, pages 371–378, 2001.