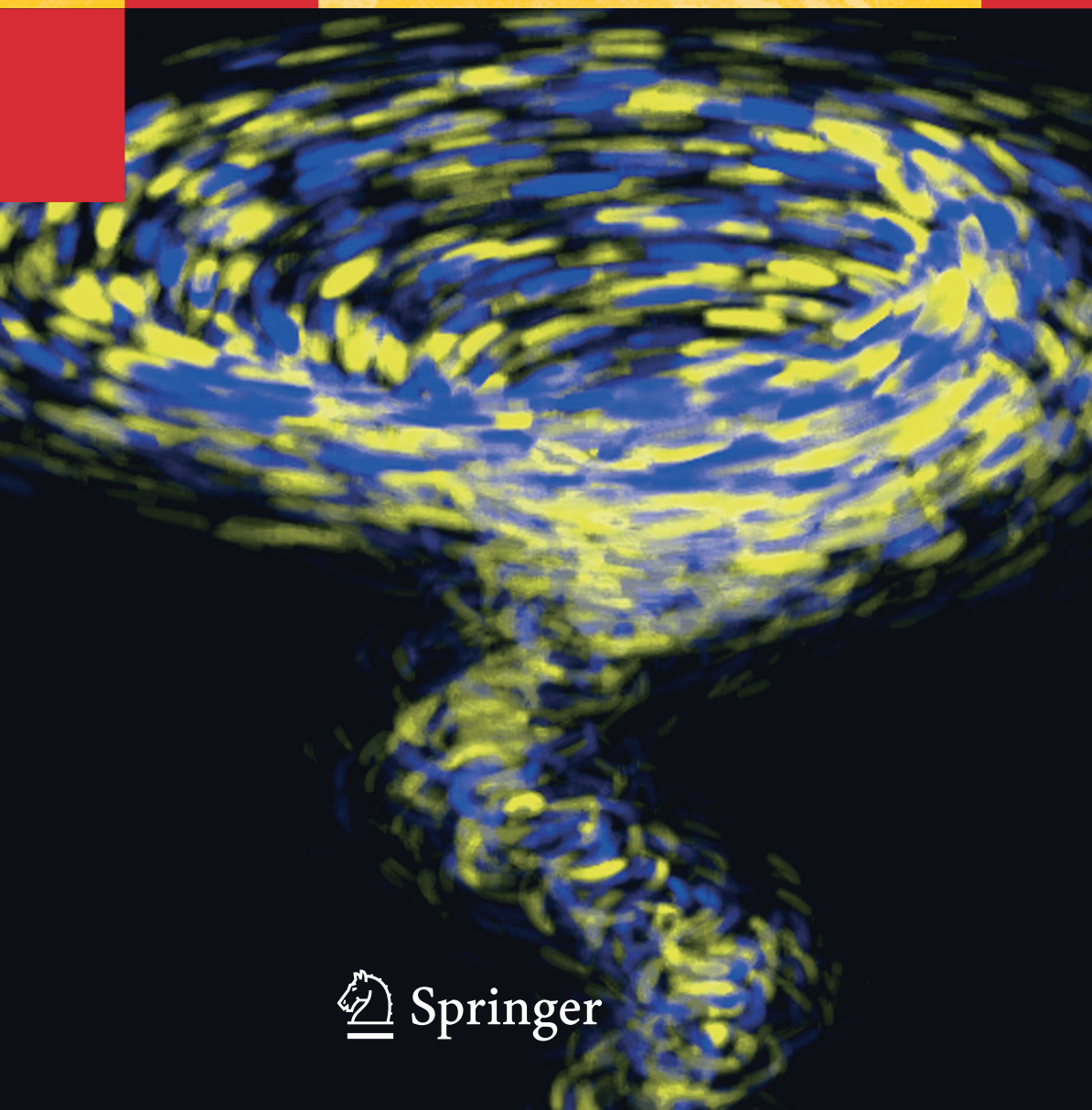
Daniel Weiskopf

GPU-Based Interactive Visualization Techniques

Mathematics + Visualization



Springer

Daniel Weiskopf

GPU-Based Interactive Visualization Techniques

With 112 Figures, 42 in Color and 11 Tables

Springer

Daniel Weiskopf
School of Computing Science
Simon Fraser University
Burnaby, BC V5A 1S6, Canada
E-Mail: weiskopf@cs.sfu.ca

Library of Congress Control Number: 2006931796

Mathematics Subject Classification: 68U05, 68W10, 76M27

ISBN-10 3-540-33262-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-332626 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media
springer.com

Typesetting by the author and SPi using a Springer LaTeX macro package
Cover picture by Daniel Weiskopf
Cover design: *design & production* GmbH, Heidelberg

Printed on acid-free paper SPIN: 11730132 46/SPi/3100 5 4 3 2 1 0

Für Elisabeth und Gerhard

Preface

Graphics processing units (GPUs) have been revolutionizing the way computer graphics and visualization are practiced. Driven by the computer-games industry and its demand for efficient hardware support for 3D graphics, GPUs have dramatically increased in performance and functionality within only a few years. Although graphics hardware is primarily designed for the fast rendering of 3D scenes, it can also be used for other types of computations. In fact, GPUs have evolved to programmable processors that can facilitate applications beyond traditional real-time 3D rendering.

This book addresses scientific visualization as one application area that significantly benefits from the use of GPUs. In general, scientific visualization has become an important tool for visual analysis in many scientific, engineering, and medical disciplines. For example, scientists and engineers regularly use visualization to interpret simulations of air or water flow in computational fluid dynamics. Another example is the medical imaging of 3D CT (computer tomography) or MRI (magnetic resonance imaging) scans. The interactive exploration of data sets is becoming increasingly more important with the growing amount and complexity of those data sets: the user – as an expert in his or her field – uses visualization as a tool to investigate the data and extract insight from it.

This book focuses on efficient visualization techniques, which are the prerequisite for interactive exploration. High performance is primarily achieved by algorithms specifically designed for GPUs and their special features. Other aspects discussed in this work include parallelization on cluster computers with several GPUs, adaptive rendering methods, multi-resolution models, and non-photorealistic rendering techniques for visualization. This book also addresses the effectiveness of visualization methods, which can be improved by taking into account perceptual aspects and user interaction. Covering both the theoretical foundations and practical implementations of algorithms, this text provides a basis to understand and reproduce modern GPU-based visualization approaches.

This work constitutes my *Habilitationsschrift*, written at the University of Stuttgart. Being a research thesis, this *Habilitationsschrift* aims at describing visualization methods on a scientific level. Therefore, the intended audience includes researchers and students of computer science who are interested in interactive visualization methods. This book may serve as a starting point to delve into the current research in GPU-based visualization. It may also serve as reading material for a course that covers scientific visualization on an advanced undergraduate or a graduate level. This work also includes a discussion of practical issues such as how algorithms are mapped to GPU programs or performance characteristics of implementations. Therefore, practitioners and software developers might also find this book interesting.

The reader is expected to have a basic background in scientific visualization, but not in GPU-based visualization methods. In addition, some familiarity with GPU programming is recommended. Although this book does not cover these background topics in detail, it nevertheless contains some introductory material on the basics of visualization and GPU programming. In particular, a wealth of references is provided to guide the reader to background reading.

How to Read This Book

This book is structured in a way that it can be read from cover to cover. However, you may also pick out some passages that are most interesting to you, and you may read through the book in non-sequential order. This section gives some hints on what topics are covered in which chapter and which parts of the book are built on other parts.

It is recommended reading the introductory Chap. 1 because it describes basic concepts used to organize the book. In particular, the abstract visualization pipeline is discussed along with a classification scheme for visualization methods (Sect. 1.1). Throughout this book, visualization techniques are related to the three main stages of the visualization pipeline, namely filtering, mapping, and rendering. This chapter also covers fundamentals of GPUs (in Sect. 1.2) that can be skipped if you are already familiar with GPU programming. In addition, Sect. 1.3 presents methods and goals of this work.

The main part of the book is organized in three large chapters on 3D scalar field visualization (Chap. 2), vector field visualization (Chap. 3), and perception-oriented and non-photorealistic rendering (Chap. 4). To a large extent, each of these chapters represents a portion of the book that can be read independently of the other main chapters. However, a few interdependencies are present as outlined below.

Chapter 2 addresses methods for direct volume visualization of 3D scalar fields. It is recommended reading the basics of volume rendering as laid out in Sects. 2.1–2.3. Sections 2.1 and 2.2 describe the underlying optical model and the volume rendering pipeline. Section 2.3 discusses basic volume rendering

methods, focusing on real-time GPU rendering. In particular, texture-based volume rendering (also called texture slicing) is described because it is the method of choice in this work. A multi-bricking approach for 3D texture-based rendering is introduced in Sect. 2.4. This approach maintains an approximately constant rendering performance for real-time applications. Sections 2.5 and 2.6 discuss advanced topics that could be read independently. Section 2.5 focuses on a number of new techniques for volume clipping that allow for complex clipping geometries. Clipping plays an important role in improving the perception of a 3D data set because it enables the user to explore otherwise hidden internal parts of the data set. Here, object-space and image-space clipping methods are compared, pre-integrated volume clipping is described, and issues of consistent volume shading are discussed. The visualization of very large, time-dependent volume data is addressed in Sect. 2.6. One element of this large-data approach is parallelization on a cluster computer with commodity-of-the-shelf GPUs. Another element is wavelet compression in combination with adaptive rendering. Chapter 2 concludes with a brief summary of the described volume rendering techniques.

Chapter 3 discusses techniques for vector field visualization, with the focus on texture-based methods. Section 3.1 is recommended as basis for this chapter because it presents the fundamental concept of particle tracing. Section 3.2 provides an overview and a classification of vector field visualization methods. This section contains an extensive list of references, serving as a good starting point to delve into state-of-the-art vector field visualization. Section 3.3 continues with a more detailed discussion of texture-based vector field visualization. It describes semi-Lagrangian texture advection and shows how advection can be used for dense, noise-based vector field visualization and sparse, dye-based visualization alike. In part, this section relies on volume rendering techniques that are described in Chap. 2 (especially Sects. 2.1–2.3). Section 3.3 is recommended as basis for the following sections of this chapter. These subsequent sections cover advanced topics and can be read independently of each other. A novel level-set advection scheme is introduced in Sect. 3.4 to overcome numerical diffusion that is inherent to semi-Lagrangian dye advection. While the methods discussed so far work in Cartesian 2D and 3D space, Sect. 3.5 addresses vector field visualization on curved surfaces, for example, on the boundary surface of an automobile model enclosed by wind flow. A hybrid object-space and image-space method is introduced to achieve an efficient, yet accurate dense flow representation. Section 3.6 describes a generic framework for the visualization of time-dependent vector fields. This framework comprises all relevant previous visualization methods and allows us to compare them on a mathematical basis. In addition, the flexibility of the framework leads to the development of novel visualization approaches. The final section of Chap. 3 summarizes the presented flow visualization methods.

Perception-oriented rendering and non-photorealistic rendering are discussed in Chap. 4. This chapter focuses on the third stage of the visualization pipeline – the rendering stage. It is recommended reading the brief

discussion of previous work in Sect. 4.1. Sections 4.2–4.5 can be read independently of each other because they cover different aspects of perception-oriented and non-photorealistic rendering. Section 4.2 addresses the influence of color on the visual perception of moving patterns. Based on an extensive review of psychophysics and psychology literature, a set of design guidelines is derived for effective animated visualization. These guidelines are especially useful for texture-based flow visualization; therefore, some background reading in Chap. 3 is recommended. Section 4.3 improves depth-perception by utilizing perception-oriented color cues. In particular, depth-dependent intensity and saturation modifications are employed. Section 4.4 introduces non-photorealistic rendering methods that improve the perception of spatial structures in complex tone-shaded illustrations. A view-dependent transparency model is proposed in Sect. 4.4.2, whereas alternative cutaway methods are described in Sect. 4.4.3. These tone-shaded illustrations are tightly connected to non-photorealistic volume rendering and volume clipping. Therefore, some background reading in Chap. 2 is recommended. Non-photorealistic halftoning approaches are explicated in Sect. 4.5; here, the focal point is frame-to-frame coherent halftoning and a generic GPU-based concept for G-buffer operations. Frame-to-frame coherent halftoning relies on texture advection as a basic technique. Therefore, background material from Sect. 3.3 is useful for understanding Sect. 4.5. Chapter 4 ends with a summary of presented rendering methods.

Following the main part outlined above, Chap. 5 concludes this book. This chapter classifies the visualization methods discussed in this book and puts them in context. The appendix contains lists of figures, tables, and color plates, as well as a bibliography, an index, and color plates.

Acknowledgments

I conducted the research for this book while I was with the Institute for Visualization and Interactive Systems (VIS) at the University of Stuttgart. I would like to thank all VIS members for the great time I had in Stuttgart and for many fruitful discussions. In particular, I thank Thomas Ertl for lots of support and advise; Joachim Diepstraten, Mike Eißele, and Sabine Iserhardt-Bauer for having been great roommates; and Matthias Hopf, Guido Reina, Ulrike Ritzmann, and Simon "Grbic" Stegmaier for their help.

I especially thank the following VIS members for successful collaborations on research that has become part of this work: Joachim Diepstraten [88, 89], Mike Eißele [108, 109], Klaus Engel [469, 470], Matthias Hopf [473, 478, 479], Stefan Röttger [356], Martin Rotard [352], Tobias Schafhitzel [481], Magnus Strengert and Marcelo Magallón [402], as well as Manfred Weiler [484]. In addition, I had the pleasure to work with many VIS members on topics that, although not directly part of this book, have influenced my research in general: Ralf Botchen [27, 468], Joachim Diepstraten [87, 90], Mike

Eißele [107, 216], Thomas Klein [216], Guido Reina [351], Dirc Rose [351], Martin Rotard [353], Tobias Schafhitzel [374, 480], Frederik Schramm [483], and Simon Stegmaier [107, 351]. Furthermore, I would like to thank the following collaborators outside VIS: Gordon Erlebacher [115, 471, 472, 473, 483], Stefan Guthe [356, 402], Bob Laramee and Helwig Hauser [240, 242], as well as Frits Post [240]. Finally, I thank Manfred Weiler for proof-reading Chaps. 1 and 2, and Nikolaus Weiskopf for help and discussions on various topics.

Financial support from the following funding organizations has made my research possible: the "Landesstiftung Baden-Württemberg" for support within the program "Eliteförderprogramm für Postdoktoranden"; the "Deutsche Forschungsgemeinschaft" (DFG) for funding the projects D4 and D8 within SFB 382 and the project C5 within SFB 627; and the University of Stuttgart for financing the project "i-modanim" within the program "self-study online".

I also would like to thank the BMW group and Roger Crawfis for providing data sets, Gerik Scheuermann for making available the image for Fig. 3.1 (d), and Martin Schulz for providing the image for Fig. 3.3. The following publishers have kindly provided the permission to use material for this book from papers (co-)authored by myself: A K Peters, Ltd. [475]; Elsevier/Academic Press [471]; the Eurographics Association [88, 89, 356, 402, 464, 481]; IEEE [108, 465, 469, 470, 472, 484]; UNION Agency/WSCG [476]; and the publisher of the SimVis 2004 proceedings (SCS Publishing House) [109].

I thank Thomas Ertl, Eduard Gröller, Kurt Rothermel, and Hans-Peter Seidel for reviewing my *Habilitationsschrift*. I also thank the Faculty 5 (Computer Science, Electrical Engineering, and Information Technology) at the University of Stuttgart for a very timely organization and pleasant handling of the *Habilitation* procedure.

I am indebted to all people at Springer who supported this book project. In particular, I thank Martin Peters (Executive Editor, Mathematics, Computational Science, and Engineering) and the editors of the book series *Mathematics and Visualization*, Gerald Farin, Hans-Christian Hege, David Hoffman, Christopher R. Johnson, Konrad Polthier, and Martin Rumpf. I also thank Ute McCrory for organizing the publishing process.

Very special thanks to Bettina A. Salzer for thorough proof-reading. This work would not have been possible without her help, patience, and love. Finally, I thank my parents Elisabeth and Gerhard for all their support.

Vancouver,
April 2006

Daniel Weiskopf

Contents

1

Introduction

Visualization has become an essential part in today's engineering, research, and business workflows. The typical amount of data that originates from numerical simulations or sensor measurements is very large and, thus, visualization is indispensable for understanding this data. The ever increasing performance of supercomputers and resolution of scanning devices is a challenge that requires us to keep on developing improved visualization methods.

This book addresses this challenge in manifold ways. Both the efficiency and effectiveness of visualization methods are improved. High efficiency is achieved by using appropriate algorithms and data structures. This work specifically considers the usage of the fast graphics processing units (GPUs) of modern graphics hardware to obtain high performance – the performance gain can be as much as two to three orders of magnitude compared to a CPU (central processing unit) implementation. The effectiveness of visualization methods is improved by taking into account perceptual aspects and an appropriate representation of the underlying data model.

Efficiency and effectiveness are not completely separated but interlinked: a highly efficient visualization method is the prerequisite for an interactive application, which, in return, leads to significantly improved effectiveness. For example, an interactive exploration may promote spatial perception by motion parallax or result in a better understanding of the large parameter space that controls data generation and visualization. Particularly interesting is real-time visualization for time-dependent data. Increased efficiency can lead to a qualitatively improved effectiveness because it might make the difference between a visualization that allows for effective user interaction and a non-interactive inefficient implementation.

The use of GPUs for high-performance visualization is motivated by several aspects. First, GPUs typically have more transistors than CPUs. For example, an NVIDIA GeForce 6800 Ultra GPU comprises 222 million transistors, whereas an Intel Pentium 4 EE (Extreme Edition) CPU contains 29 million transistors for the processor core and 149 million transistors for the L2 (level 2) cache [419]. Second, the speed of GPUs and their number of transis-

tors increase faster than described by Moore's law for CPUs. Third, graphics hardware has wide data paths to its memory, and efficient bilinear and trilinear reconstruction of texture data is available. Fourth, the programming model for GPUs is rather restricted and results in highly parallel execution and deep pipelining. Finally, a transfer of visualization results to the graphics board for final display is superfluous for GPU-based implementations.

1.1 Visualization Pipeline and Classification of Visualization Methods

An abstract *visualization pipeline* is used throughout this book to structure the large variety of visualization methods. This pipeline approximately follows the description by Haber and McNabb [149] – up to some slight modifications. Figure 1.1 sketches the elements of the visualization pipeline.

Input to the visualization pipeline is acquired from a data source, such as a numerical simulation, a measurement of physical data, or a database. This *raw data* is transformed by the *filtering* stage of the visualization pipeline into abstract *visualization data*. Typical filtering operations are denoising by convolution with a filter kernel or data enhancement by segmentation. The next stage is visualization *mapping*, which constructs a *renderable representation* from visualization data. The renderable representation has extension in space and time, and it contains attribute fields that may comprise geometry, color, transparency, reflectance, and surface texture. A typical example for a

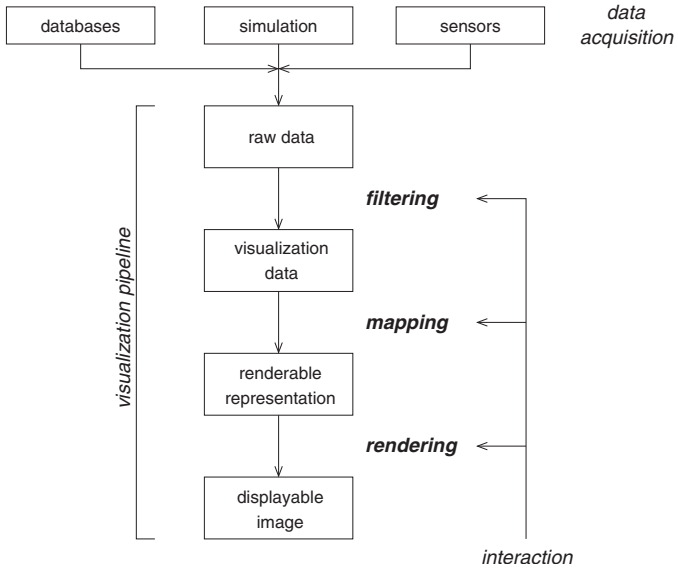


Fig. 1.1. Abstract visualization pipeline

mapping operation is the application of a color table that assigns colors to input data values. Finally, the *rendering* stage generates a displayable image from the renderable representation. Typical rendering operations include view transformations, scene illumination, and shading.

For some visualization algorithms, it may be difficult to find a clear and unique assignment of algorithmic elements to the filtering, mapping, or rendering stages. GPU-based methods, in particular, often require a reordering of the conceptual visualization pipeline to achieve an efficient implementation. Examples for such a reordering are given for corresponding visualization methods when their details are discussed later in this work. Nevertheless, the concept of the visualization pipeline has proven to be useful in classifying and comparing different visualization methods. For example, this scheme has been successfully used as a basis for teaching visualization courses (see the article by Rotard, Weiskopf, and Ertl [352]). Therefore, all visualization techniques of this book are analyzed by taking into account the concept of the visualization pipeline.

As pointed out in Fig. 1.1, a user may potentially interact with all three stages of the visualization pipeline. Interaction plays a crucial role throughout this work because the large parameter space that controls data acquisition and visualization can only be explored interactively. As a prerequisite, visualization techniques need to be efficient to allow for real-time application.

In addition to the structure of the visualization pipeline, a more detailed taxonomy is useful for classifying visualization methods. Following Bergeron and Grinstein [17] and Brodlie [31], the dimensionality of independent variables (on the domain) and the dimensionality of dependent variables (i.e., data type) are important categories. Therefore, visualization methods are ordered according to their data type, starting with scalar data in Chap. 2. Vector fields, which are more complex than scalar fields, are discussed in Chap. 3. In addition, the dimensionality of the domain plays an important role in substructuring this chapter on vector field visualization. Chapters 2 and 3 cover all steps of the visualization pipeline for the specific cases of scalar and vector fields, whereas Chap. 4 focuses on the rendering stage. This chapter discusses perception-oriented and non-photorealistic rendering methods that could be applied to scalar field and vector field visualization, but also to a much wider range of visualization applications.

A third category that can be used to classify visualization methods is the type of data structure that holds the data set. The data structure is important from an algorithmic point of view and therefore influences the design of visualization algorithms. Throughout this book, visualization techniques for different data structures are discussed. Usually, data is represented by a set of data points located in the domain space. The data structure forms a *grid* if a connectivity between data points is established. In contrast, *scattered data* does not have such a connectivity. We can distinguish two classes of grids: *structured* and *unstructured* grids. Structured grids have an implicitly given connectivity, whereas unstructured grids need their connectivity explic-

itly stored. Examples of structured grids are *Cartesian*, *uniform*, *rectilinear*, or *curvilinear* grids. Cartesian, uniform, and rectilinear grids have all their cells aligned with the coordinate axes of the domain. A Cartesian grid has the same overall grid spacing regardless of direction. A uniform grid has an equidistant spacing along each of the directions, although the spacing is independently chosen for the different directions. A rectilinear grid has varying cell sizes. A curvilinear grid exhibits the most flexibility by allowing for deformed cells, provided the topological structure is still regular. A typical example of an unstructured grid is a grid consisting of simplical cells, i.e., a tetrahedral grid in the 3D case. More details on these basic grid types and other, more advanced grid structures are given in the book by Schroeder et al. [382].

1.2 GPU Rendering Pipeline

This book discusses efficient visualization algorithms for GPUs. The basic properties of GPUs, their programming model, and the underlying rendering pipeline are outlined in this section. The stages of the GPU rendering pipeline are only briefly reviewed; detailed background information on the traditional fixed-function pipeline can be found in a textbook by Foley et al. [122] or the OpenGL programming guide [496]. Additional information on using programmable graphics hardware is given in Cg manuals [118, 312] and documentations of the OpenGL shading language [207] and Direct3D [288], which is part of DirectX.

Figure 1.2 sketches the GPU rendering pipeline. A 3D graphics application communicates with graphics hardware via a 3D API (application programming interface). Typically, either OpenGL or Direct3D are used as 3D API. 3D scenes are usually described by a boundary representation (BRep) formed by a collection of primitives. In most cases, primitives are triangles defined by vertices. Vertex information consists of geometric position and further attributes such as normal vector, texture coordinates, color, or opacity.

Primitives are sent from CPU to GPU as a stream of vertices, which are then transformed by the *vertex processor*. In the traditional fixed-function graphics pipeline, vertex processing accomplishes *transform* and *lighting* operations. A vertex of the input geometry is transformed from its original object coordinates into subsequent world, eye, and normalized clip coordinates by multiplication with respective matrices for model, view, and normalization transformations. Matrix formulations of these affine or projective transformations are adequate because all computations are based on homogeneous coordinates. Lighting is usually based on the Blinn-Phong model and performed with respect to eye coordinates. GPUs allow the programmer to replace these fixed-function vertex operations by a flexible vertex program (OpenGL lingo) or vertex shader (Direct3D lingo).

Primitives are converted into fragments after vertex processing. Today's GPUs exclusively process triangles at these stages of the pipeline; i.e., prim-

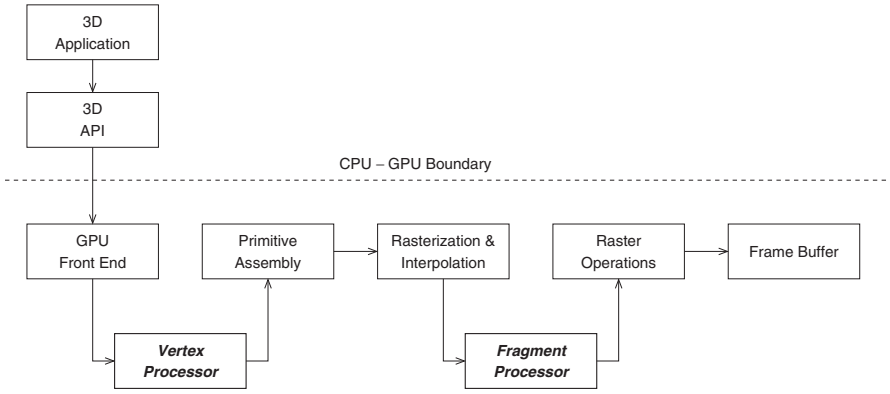


Fig. 1.2. GPU graphics pipeline (inspired by [312])

itive and triangle are synonymous in this context. During *primitive assembly* and *rasterization* the following operations are executed. Individual vertices of the vertex pipeline are organized into primitives, and primitives are clipped against the viewing frustum. At the transition from the vertex pipeline to the fragment pipeline, rasterization employs scanline conversion to generate the fragments covered by an input triangle. At the same time, vertex attributes are *interpolated* for fragments that lie inside the input triangle. Position coordinates are transformed from normalized clip space into normalized device coordinates by a homogeneous division by the w clip coordinate. Also, the viewport transformation to window coordinates is applied. Primitive assembly, rasterization, and interpolation are fixed and cannot be programmed by the user.

The subsequent *fragment processor* modifies fragment attributes and applies textures. Texture lookup can make use of efficient built-in bilinear or trilinear texture interpolation. Analogously to the vertex processor, fragment processing can either be used as a traditional fixed-function element or freely specified in the form of a fragment program (OpenGL lingo) or pixel shader (Direct3D lingo).

The final raster operations combine the current fragment with pixel information given at the same position in the frame buffer: first, the alpha, stencil, and depth tests (the latter is also called z test) are executed; then, if a fragment passes all tests, it may be directly written into the frame buffer or composited with frame buffer information by blending.

1.2.1 Programming Model

GPU-based visualization methods utilize the programmability of the vertex processor and fragment processor stages. Typical operations for the vertex processor comprise the calculation of quantities that can later be interpolated

linearly via scanline conversion. For example, texture coordinates can be generated to attach a 3D vector field stored in a 3D texture to a surface that cuts through the vector field. Much more important, however, is the fragment processor because it allows us to compute information on a per-fragment level. Moreover, an efficient access to texture data is only possible at this stage. Therefore, the visualization methods of this book make extensive use of pixel shaders.

Fragment processing can be considered as a streaming model of computation [38]. The streaming model extends the SIMD (single instruction, multiple data) concept that was used as abstraction of the OpenGL architecture [324], where each rendering pass implements a SIMD instruction that performs a basic arithmetic operation and updates the frame buffer atomically. The stream programming model represents computational locality that is not present in the SIMD model. The key elements are streams and kernels. A stream consists of a set of data elements for which similar computations are executed. A streaming processor applies a computing kernel to all elements of an input stream and writes the results into an output stream. Dally et al. [70] and Buck et al. [38] discuss how the stream model supports programs with high arithmetic and computational intensity (i.e., a large ratio of arithmetic operations to memory bandwidth). Figure 1.3 depicts the basic execution model of a GPU that applies to fragment and vertex processing alike. A graphical input element (e.g., a vertex or a fragment) is transmitted through read-only input registers into the shader program. The result of shader execution is written to output registers. The shader program has read and write access to temporary registers, and read-only access to textures and constants. Therefore, a shader can be regarded as a stream kernel that is applied to stream data held in input registers or textures. Please note that vertex programs used to have no access to texture; however, with Shader Model 3 compliant GPUs (e.g., NVIDIA GeForce 6 and GeForce 7 series), texture access is also available in vertex programs.

Although the streaming model slightly restricts the broader functionality of GPUs, it is well-suited for most visualization methods of this book. Efficient visualization techniques thrive on a good mapping to the streaming model. For example, such a mapping has to achieve a uniform kernel structure with a high degree of parallelism, no branching, and little or no conditional expressions. A regular data access to textures is highly advantageous to utilize optimized data paths to internal memory. If possible, efficient built-in bilinear or trilinear interpolation should be exploited. The stream output often needs to be re-used as input to a subsequent stream computation. In this case, the output register is directly transferred into a texture by using the efficient render-to-texture functionality of GPUs.

A number of specific GPU properties should be taken into account. First, the accuracy of GPUs is usually limited and might vary along the rendering pipeline. For example, color channels in the frame buffer or in textures have a typical resolution of eight bits, whereas fragment processing may take place

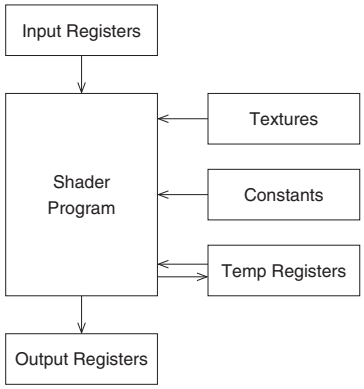


Fig. 1.3. Programming model for current GPUs (after [38]). A shader program processes a single input element and writes the result to an output register

at higher precision. Even floating-point accuracy within textures and fragment processing, which is provided by modern GPUs, is not comparable to double-precision numbers, which are available on CPUs. Second, the number of instructions might be limited. Therefore, the algorithms have to be designed in a way that allows a concise implementation – sometimes at the cost of accuracy – or a less efficient multi-pass rendering approach has to be pursued. Similarly, the number of indirection steps in fragment processing (i.e., dependent texture lookups) may be restricted. Third, an efficient abortion of a fragment program (`texkill` command) is not possible on many GPUs. Therefore, the early z test is often (mis)used to emulate such a program abortion. Fourth, GPUs can simultaneously apply numerical operations to four different color channels (red, green, blue, alpha) of a register, which can be used for an efficient implementation. Fifth, a read-back of information from GPU to main memory should be avoided because the bandwidth from GPU to CPU can be significantly smaller than the bandwidth from CPU to GPU. Finally, a simultaneous read and write access to textures is not specified; e.g., an in-place replacement of texel information is not feasible. This problem can be overcome by a ping-pong rendering scheme: two copies of a texture are held, one for the read access to "old" information (regular texture read) and the other one for writing "new" information (with a render-to-texture operation); after each computational step, both textures are swapped and their roles are exchanged, i.e., a ring buffer with two elements is employed.

Although GPUs have a potential to increase visualization performance by more than one order of magnitude, an inappropriate implementation or algorithm could decrease the speed of a GPU version significantly, compared to a reference CPU version. Therefore, the above issues and restrictions have to be carefully taken into account. Specific problems and their solutions are discussed throughout this work in the context of respective visualization algorithms.

1.2.2 APIs and Effect Files

Another issue is the choice of API for GPU programming. High-level scene graph APIs, such as OpenInventor, OpenGL Performer, OpenSceneGraph, or OpenSG, are not considered in this book because they anyway employ low-level APIs for GPU configuration and actual rendering. Therefore, only the widespread low-level APIs OpenGL and Direct3D are used in this work.

GPU programs can be developed on assembler level or with high-level shading languages. Typical assembly-like programming environments are ARB vertex programs and ARB fragment programs for OpenGL and vertex shader and pixel shader programs for Direct3D. The advantage of assembler-level shaders is the direct control of shader instructions, which allows for hand-coded optimization. The disadvantages are a time-consuming and error-prone shader development and program codes that are difficult to read and maintain. These problems are overcome by high-level shading languages, such as the OpenGL shading language [207], Direct3D's HLSL [288], or NVIDIA's Cg [118, 312]. These high-level shading languages for GPUs are all very similarly structured, oriented along the C programming style, and strongly influenced by the Renderman shading language [152]. Old GPU programming concepts such as NVIDIA's register combiners or texture shaders [210] do not play a role in this book because they are vendor-specific and inflexible. However, references to older publications based on such GPU programming models are sometimes included in the text.

In addition to actual vertex and pixel shaders, states of the graphics pipeline strongly influence rendering results, i.e., GPU programming is more than shader development. Traditionally, different parts of the program source code (for example, in the C++ code) are responsible for setting graphics states. *Effect files* (FX files) were introduced in Direct3D 8 and further enhanced in Direct3D 9 [288] to simplify GPU programming. They encapsulate the whole setup and programming of the graphics pipeline within a so-called *effect*. An effect is described by a character string or an external text file and thus this concept is mainly referred to as effect file. The source code responsible for the GPU programming is decoupled from the main program code, which deals with CPU programming. In this way, a better maintainability and readability of the source code for both parts is achieved.

1.3 Methods and Goals

The goal of this book is the development of methods that improve both the efficiency and effectiveness of visualization. Efficiency mainly concerns technical aspects of algorithms, data structures, and implementations, whereas effectiveness is strongly affected by perceptual issues related to the human observer and domain knowledge for a specific application.

A wide spectrum of subgoals requires an equally broad and interdisciplinary set of methods. Typical methods from practical computer science are concerned with the use of efficient algorithms and memory-friendly data structures with fast access. Due to the restricted programming model of GPUs, specific algorithms that are efficiently supported by GPUs are developed. Algorithms are typically validated by running tests on their implementation, along with performance measurements. Actual implementation on GPUs is indispensable to verify these algorithms because GPU programming is subject to various subtle pitfalls. Moreover, visualization methods are discussed in the context of the generic visualization pipeline to show differences and analogies. Parallelization on distributed-memory architectures, such as cluster computers, is another computer-science approach used in this book.

Mathematical methods are equally important to achieve efficient visualization techniques. For example, appropriate numerical methods have to be employed for differential equations and level-sets, wavelets are used for multiresolution modeling and compression, and reconstruction and filter theory plays a significant role in analyzing convolution-based vector field visualization.

The last major aspect takes into account human perception and cognition because visualization techniques have to target comprehension by a human observer. This book thrives on knowledge from fields like psychophysics, psychology, and neurophysiology to achieve perception-oriented rendering. Literature in these fields is analyzed to find formalized methods that can be transferred into a computer program. Likewise, well-established experience from artists, designers, and illustrators is condensed into perception-guided and non-photorealistic rendering algorithms.

The generic goals of efficiency and effectiveness are pursued for specific but widely useful applications: the visualization of 3D scalar fields and vector fields in 2D and 3D. Interactivity plays a crucial role in all applications because it greatly enhances their effectiveness. For example, interactivity allows a user to explore the large parameter space of visualization and data generation, it promotes spatial perception by motion parallax, and naturally represents temporal changes of time-dependent data. Real-time capable visualization is a requirement achieved by efficient GPU techniques that accelerate the complete visualization pipeline.

2

Visualization of 3D Scalar Fields

Volume rendering is a widely used technique in many fields of application, ranging from direct volume visualization of scalar fields for engineering and sciences to medical imaging and finally to the realistic rendering of clouds or other gaseous phenomena in visual simulations and virtual environments. In recent years, texture-based volume rendering on consumer graphics hardware has become a popular approach for direct volume visualization. The performance and functionality of GPUs have been increasing rapidly, while their prices have been kept attractive even for low-cost PCs.

Volume rendering targets the visualization of 3D scalar fields. A time-dependent scalar field, in general, is a map

$$s : M \times I \longrightarrow \mathbb{R} \, ,$$

where M is an m-dimensional manifold with boundary and $I \subset \mathbb{R}$ is an open interval of real numbers. An element $t \in I$ serves as a description for time, $x \in M$ is a position in space, and the result of s yields the associated scalar value.

For the more specific case of volume rendering, data is given on a flat 3D manifold. A corresponding 3D time-dependent scalar field is described by

$$s : \Omega \times I \longrightarrow \mathbb{R} \, , \quad (\mathbf{x}, t) \longmapsto s(\mathbf{x}, t) \, .$$

This scalar field is defined on the 3D Euclidean space $\Omega \subset \mathbb{R}^3$ and depends on time $t \in I$. For a stationary 3D scalar field, the time-dependency is obsolete:

$$s : \Omega \longrightarrow \mathbb{R} \, , \quad \mathbf{x} \longmapsto s(\mathbf{x}) \, .$$

This chapter focuses on the visualization of scalar fields without explicit time-dependency: either the input data set is already stationary or just a single time step of a time-dependent data set is considered at a time.

The basic idea of direct volume visualization is to simultaneously show all important aspects of a data set in a single image. Semi-transparent rendering has to be applied to represent a full 3D data set on a 2D image plane.

Section 2.1 describes the prevalent optical model used for semi-transparent volume rendering. Essentially, viewing rays starting at the camera are traversed through the data set. Color contributions are accumulated while stepping along a viewing ray. This image synthesis process can be structured via the volume rendering pipeline, which consists of the stages data traversal, interpolation, gradient computation, classification, shading, and compositing (see Sect. 2.2).

The volume rendering pipeline can benefit from direct GPU support. Section 2.3 describes basic volume rendering methods, focusing on real-time GPU rendering. To a large extent, this chapter addresses the visualization of data sets given on uniform grids. This type of grid has a natural representation via 3D textures or stacks of 2D textures. Both the scalar data set and its gradients can be held in these textures. An advantage of a GPU is that interpolation on textures is efficient due to built-in hardware support. Section 2.3 also discusses alternative methods that support unstructured grids, especially tetrahedral grids. Furthermore, Sect. 2.3 shows how data traversal can be implemented using graphics hardware and how classification, shading, and compositing can be handled by fragment processing.

The stages of the volume rendering pipeline can be related to the steps of the more abstract visualization pipeline. The classification stage can be identified as the mapping step, whereas shading and compositing can be considered as rendering step. Direct volume visualization exemplifies the fact that the visualization pipeline should be regarded as a conceptional pipeline. The actual implementation shows that a direct combination of the three steps may be more efficient than a clear separation with a subsequent execution of the steps of the visualization pipeline. For volume rendering, the order of operations is significantly modified – the roles of the outer processing of the visualization pipeline (the filtering, mapping, and rendering steps) and the inner loop (stepping along a viewing ray) are exchanged.

This chapter particularly focuses on two aspects of direct volume visualization. The first one is volume clipping, which is discussed in detail in Sect. 2.5. Clipping provides a means of selecting regions of the data set on a purely geometric basis and plays a decisive role in understanding 3D volumetric data sets because it allows the user to cut away selected parts of the volume based on the position of voxels in the data set. Clipping reduces the domain of the data set to be visualized; therefore, clipping can be regarded as an example of a selection process (with respect to the domain), which is part of the filtering step of the visualization pipeline. From an alternative point of view, one may associate clipping with the mapping step (in the form of a space-dependent mapping to opacities) or the rendering step (as a clipping of the renderable geometry). Clipping shows that the assignment to the different steps of the visualization pipeline is not always unique. In this chapter, however, clipping is considered as part of the filtering step. The second important aspect covered in this chapter is the efficient rendering of large volume data sets. Section 2.6 addresses large-data visualization by applying compression techniques, adap-

tive rendering, and parallel processing on GPU clusters. A complementary multi-bricking approach leads to constant frame rates for volume rendering with 3D textures, independent of the viewing positions (see Sect. 2.4). Both volume clipping and large-data visualization rely on efficient texture-based volume rendering as a basic technique (see Sect. 2.3).

2.1 Optical Model for Volume Rendering

Direct volume visualization can be derived from optical models for volume rendering and volume shading. Here, a brief overview of a widely used approach is given. A detailed presentation of the underlying physical and mathematical models can be found in the review articles by Hege et al. [160] or Max [275]. This chapter follows the terminology used by Max.

By ignoring scattering, the generic equation of transfer for light can be reduced to the simpler emission-absorption model or density-emitter model [362]. The respective volume rendering equation can be formulated as the differential equation

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = g(t) - \tau(t)I(t) \; . \tag{2.1}$$

The amount of radiative energy is described by radiance $I(t)$, and its derivative with respect to the length parameter t is taken along the direction of light flow. The source term $g(t)$ describes the emission of light from the participating medium (and later, through reflection as well). The extinction coefficient $\tau(t)$ defines the rate that light is attenuated by the medium. The optical properties are determined by a transfer function that assigns a value for the extinction coefficient and the source term to each sample point of the data set. Radiance and optical properties are written as scalars, which is appropriate for luminance-based gray-scale images. A wavelength-dependent behavior or multiple wavelength bands (for example, red, green, and blue) can be taken into account by computing these quantities for each wavelength separately.

The volume rendering equation (2.1) can be integrated to compute the radiance that leaves the volume and finally arrives at the eye point. Integrating from the initial point $t = t_0$ to the end point $t = D$ along the light direction results in

$$I(D) = I_0 \, e^{-\int_{t_0}^{D} \tau(t') \, \mathrm{d}t'} + \int_{t_0}^{D} g(t) \, e^{-\int_{t}^{D} \tau(t') \, \mathrm{d}t'} \, \mathrm{d}t \; .$$

The background light that enters the volume at the position $t = t_0$ is denoted by I_0. The radiance that leaves the volume at $t = D$ is described by $I(D)$. With the definition of transparency,

$$T(t) = e^{-\int_{t}^{D} \tau(t') \, \mathrm{d}t'} \; ,$$

one obtains the volume rendering integral

$$I(D) = I_0\, T(t_0) + \int_{t_0}^{D} g(t) T(t)\, \mathrm{d}t \; . \tag{2.2}$$

So far, the optical model is restricted to emission and absorption. The visual realism of volume rendering can be substantially increased by considering scattering of light. A simple and widely used volume illumination model implements single scattering by assuming that light from an external light source unimpededly reaches the scattering location. Light attenuation is neglected on the way from the light source to the scattering point, which is similar to neglecting shadowing in surface graphics. The result of single scattering can be computed similarly to local illumination models known from surface rendering, such as the Phong or the Blinn-Phong models. The gradient of the scalar field is used as the normal vector in local illumination because the gradient is identical to the normal vector of an isosurface. In this way, volume illumination essentially imitates the effect of lighting isosurfaces. The source term of the volume rendering integral (2.2) can be extended to include local illumination:

$$g(\mathbf{x}, \omega) = E(\mathbf{x}) + S(\mathbf{x}, \omega) \; . \tag{2.3}$$

Here, the source term is given with respect to position $\mathbf{x}$, which directly corresponds to the length parameter t along the light ray. The non-directional emissivity $E(\mathbf{x})$ is identical to the source term in the pure emission-absorption model. The additional scattering term $S(\mathbf{x}, \omega)$ depends on the position $\mathbf{x}$, the direction of the reflected light, ω, and the properties of the light source.

The integral in (2.2) is typically approximated by a Riemann sum over n equidistant segments of length $\Delta x = (D - t_0)/n$. The approximation yields

$$I(D) \approx I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_i \; , \tag{2.4}$$

where

$$t_i = e^{-\tau(i\Delta x + t_0)\Delta x} \tag{2.5}$$

is the transparency of the ith segment and

$$g_i = g(i\Delta x + t_0)\Delta x \tag{2.6}$$

is the source term for the ith segment. Note that both the discretized transparency t_i and source term g_i depend on the segment length Δx. Opacities $\alpha_i = 1 - t_i$ are often used instead of transparencies t_i. When the RGB color model is applied, the source term g_i is described by its RGB values. The emissive RGB part is denoted C_i. Therefore, source and opacity terms can be combined in the form of an RGBA contribution. These terms have to be adapted if the sampling distance Δx is changed.

Although the emission-absorption model is predominant in real-time volume graphics, it should be pointed out that there exist more advanced rendering algorithms that strive for a higher degree of realism by including further elements of global illumination. Background information on the physical models for advanced global illumination can be found in the physics literature, for example, in a book by Chrandrasekhar [53]. Light transport can be regarded as a special case of a generic transport mechanism described by the Boltzmann equation. Mathematical details of the Boltzmann equation are discussed, for example, by Duderstadt and Martin [100] or Case and Zweifel [47]. Transport theory can be directly applied to volume visualization, as shown by Krueger [230, 231], and to image synthesis in general, as shown by Arvo and Kirk [4].

There are many previous papers on advanced global illumination specifically designed for participating volumetric media. Due to their complexity, most of these global illumination algorithms are implemented on CPUs and do not facilitate interactive rendering. Some examples are volume photon mapping [186, 95], the zonal method for radiosity [360], the use of spherical harmonics [199], or a method for scattering in layered surfaces [151]. Global illumination and physics-based light transport for computer graphics in general are presented in both breadth and detail in textbooks by Pharr and Humphreys [333] and Dutré et al. [103]. The overview by Cerezo et al. [51] focuses on state-of-the-art rendering techniques for participating media.

2.2 Volume Rendering Pipeline

The evaluation of the emission-absorption model for a given data set can be split into several subsequent stages of the volume rendering pipeline. This section reviews the volume rendering pipeline only briefly; a more detailed description is given by Pfister [330]. The following stages are commonly found in volume rendering techniques: (a) data traversal, (b) interpolation, (c) gradient computation, (d) classification, (e) shading, and (f) compositing.

During data traversal, resampling positions are chosen throughout the volume. Some kind of interpolation scheme is applied at these positions to reconstruct the data set at locations that differ from grid points. Typical filters are nearest-neighbor interpolation or variations of linear filtering. Trilinear interpolation is most common for uniform grids and is also used in most of the methods presented in this book.

The gradient of a discretized volumetric data set is typically approximated by using discrete gradient filters. Many gradient filters are extensions of 2D edge filters from image processing, e.g., the Sobel operator. Central differences are a popular way of computing the partial derivatives for the gradient because they require only a small number of numerical operations.

Classification maps properties of the data set to optical properties for the volume rendering integral (2.2), i.e., it represents the relationship between

data and the source term g and the extinction coefficient τ. The classification typically assigns the discretized optical properties C_i and α_i, which are combined as RGBA values. Pre-classification first maps grid points of the data set to optical properties and afterwards applies the interpolation scheme to the RGBA values. Post-classification, in contrast, first evaluates the interpolation scheme for the input data set and then assigns the corresponding optical properties.

The mapping to optical properties is represented by a transfer function. The transfer function typically depends on the scalar value s, which is given at position $\mathbf{x}$ by the term $s(\mathbf{x})$. This leads to a slightly modified description of the source term via $\tilde{g}$:

$$g(\mathbf{x}) = \tilde{g}(s(\mathbf{x})) \; .$$

Similarly, the extinction coefficient can be written as $\tilde{\tau}$ according to

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

An analogous description can be applied for the RGBA transfer function used in the discrete approach. Additional parameters are often included in transfer functions, e.g., gradient magnitude or even higher-order derivatives [212, 217] in multi-dimensional transfer functions. Volume shading can be incorporated into transfer functions by adding an illumination term, as shown in (2.3). The design of useful transfer functions, in general, is a crucial aspect of volume visualization that attracts a lot of attention [331].

The discretized volume rendering integral (2.4) can be iteratively computed by compositing. Two types of compositing schemes are most common: front-to-back and back-to-front compositing. Front-to-back compositing is used for stepping along viewing rays from the eye point into the volume. The front-to-back iteration equations are

$$\begin{aligned} C_{\mathrm{dst}} &\leftarrow C_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) C_{\mathrm{src}} \; , \\ \alpha_{\mathrm{dst}} &\leftarrow \alpha_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) \alpha_{\mathrm{src}} \; . \end{aligned}$$

Color is accumulated in C_{dst} and opacity is accumulated in α_{dst}. The subscript $_{\mathrm{dst}}$ is an abbreviation for "destination", denoting a variable that can be modified by an update operation. The transfer function assigns the color contribution C_{src} and opacity α_{src} as a representation of the optical properties at the current location on the ray. The subscript $_{\mathrm{src}}$ is an abbreviation for "source". The same terminology of "source" and "destination" is used by OpenGL in the context of frame-buffer blending. In fact, the front-to-back iteration equation can be implemented by blending, as discussed in Sect. 2.3. The values C_{dst} and α_{dst} are initialized with zero before ray traversal. The iteration ends once the ray has left the volume. Then, background light can be added, corresponding to the term I_0 in (2.2) and (2.4). Finally, C_{dst} contains the color that is transported to the eye.

Back-to-front compositing is an alternative scheme. Here, a viewing ray is traversed from the backside of the volume to the eye. The corresponding iteration equation is

$$C_{\mathrm{dst}} \leftarrow (1 - \alpha_{\mathrm{src}})C_{\mathrm{dst}} + C_{\mathrm{src}} \; .$$

Again, color is accumulated in C_{dst}, and C_{src} and opacity α_{src} are assigned by the transfer function. Background light is used to initialize C_{dst}. Note that opacity does not need to be accumulated to compute the final color.

In addition to these compositing schemes, a number of alternative approaches might be used. For example, maximum intensity projection (MIP) is useful in some medical imaging applications. Sometimes, the simplified emission-only or absorption-only models are applied, which only consider the emission term or the absorption term, respectively. The MIP, emission-only, and absorption-only schemes share the advantage of being order-independent: their compositing equations are commutative and therefore do not require any particular traversal order. Thus, they do not need any kind of spatial sorting. In contrast, the emission-absorption model is order-dependent and requires spatial sorting, e.g., according to a front-to-back or back-to-front scheme. In what follows, this book focuses on the compositing schemes for the emission-absorption model.

Although the volume rendering pipeline provides the usual way of describing the structure of a volume rendering algorithm, the stages of this pipeline can be identified with steps of the generic visualization pipeline: classification corresponds to the mapping step, whereas shading and compositing implement the rendering step. The other stages (data traversal, interpolation, and gradient computation) deal with internal data handling and serve as input for the mapping and rendering steps. Since these data handling stages do not provide any non-trivial filtering element, they are not considered as part of the filter step of the visualization pipeline, but are rather included in the respective mapping and rendering steps.

2.3 Volume Rendering Approaches

Several ways of implementing the generic volume rendering pipeline are discussed in this section. Volume rendering methods primarily differ in the way the volume data set is traversed. In particular, methods can be classified either as image-order approaches or as object-order approaches. Object-order methods traverse the 3D volume in its object space and project the volumetric information onto the image plane. Here, this book focuses on 2D and 3D texture slicing, which are popular object-order approaches directly supported by graphics hardware (see Sects. 2.3.1 and 2.3.2).

In contrast, image-order methods use the 2D image plane as starting point for data traversal: the image is scanned and the result of volume rendering is

computed for each pixel on the image plane. Ray casting (Sect. 2.3.3) is the most prominent example of image-order volume rendering. There is a large body of previous work on CPU-based ray casting, but this section primarily focuses on the recent developments in GPU-based ray casting.

The subsequent sections cover alternative object-order methods: shear-warp factorization in Sect. 2.3.4, splatting in Sect. 2.3.5, and cell projection in Sect. 2.3.6. Furthermore, pre-integration is discussed in Sect. 2.3.7 as a means of improving most of the previously mentioned rendering methods. Finally, Sect. 2.3.8 briefly presents some variations and extensions of volume rendering methods.

In general, this book focuses on visualization methods for Cartesian or uniform grids. Therefore, volume rendering techniques for these types of grids play a dominant role in this chapter. Nevertheless, alternative methods for other grid structures, especially tetrahedral grids, are also discussed. The reader is referred to the following detailed descriptions for information on supported grid types.

2.3.1 3D Texture Slicing

Texture slicing is a prominent example of an object-order approach for GPU volume rendering. The specific example of 3D texture slicing – also called 3D texture-based volume rendering – makes use of image-aligned slices [1, 41, 68]. These view-aligned slices can be considered as 2D proxy geometry for the actual 3D volume that has to be rendered. The data set itself is stored in a 3D texture that represents volume data given on a Cartesian or uniform grid. An advantage of 3D texture slicing is its good support by graphics hardware, which leads to efficient volume rendering. The stages of the volume rendering pipeline are now described in the context of 3D texture slicing.

For data traversal, view-aligned planar polygons are rendered. Rasterization generates fragments on the image-aligned slices, producing sampling positions throughout the volume. Figure 2.1 illustrates the layout of the view-aligned slices. The view-dependent slice polygons are generated by the CPU and clipped against the bounding box of the volume. Sampling locations in the volume are addressed by 3D texture coordinates that are attached to the vertices of the slice polygons. Trilinear interpolation is a built-in feature of 3D texture mapping on GPUs, i.e., the interpolation scheme is directly and efficiently supported.

Gradients can be pre-computed for each grid point of the data set and stored along with the scalar data. For example, the RGB channels can be used for the gradient and the alpha channel for the scalar value [485]. Data sets with pre-computed gradients need four times the memory of the original scalar data, which can be a problem because texture memory is a limited resource on most, if not all, GPUs. Alternatively, gradients can be computed on-the-fly during rasterization. Here, only scalar data has to be stored in texture

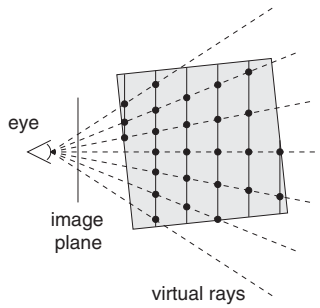


Fig. 2.1. 3D texture slicing. View-aligned slices are rendered, leading to sampling positions marked by dots. The dashed rays indicate onto which pixels the volume samples are projected

memory, but the evaluation of the gradient needs more numerical operations and texture accesses than the approach of pre-computed gradients.

Post-classification is implemented by a transfer function in the form of a color table represented by a texture. During rasterization, the interpolated scalar value is interpreted as the texture coordinate for a dependent texture lookup in the transfer function. The dimensionality of the transfer function determines the dimensionality of the dependent texture, i.e., multi-dimensional transfer functions with up to three different parameters are directly supported. Pre-classification can be implemented by paletted textures, where the index that maps to the color palette is stored in the data texture and the color palette itself represents a 1D transfer function. The shading stage is well-supported by programmable GPUs because typical illumination models such as the Blinn-Phong model can be evaluated on a per-fragment level if the gradient information is available. Even on older graphics hardware, however, volume shading is possible. For example, Westermann and Ertl [485] introduced ambient and diffuse illumination for texture-based isosurfaces. This approach could be extended to semi-transparent volume rendering [284, 285].

Compositing is typically implemented by alpha blending with the frame buffer. Both front-to-back and back-to-front strategies with their respective compositing equations are supported by alpha blending. Accumulated opacities have to be stored for front-to-back compositing, i.e., an RGBA frame buffer is required. Current graphics hardware is often restricted to 8-bit resolution for the RGBA channels of the frame buffer. Quantization artifacts can therefore occur when opacity is close to zero; see Bitter et al. [22] for a thorough discussion of quantization effects. Texture formats with 16-bit or 32-bit resolution can be used to overcome this accuracy problem [110]; here, compositing is performed by ping-pong rendering with these high-resolution textures.

2.3.2 2D Texture Slicing

2D texture slicing is an object-order approach tightly related to 3D texture slicing. The main difference is that 2D slicing employs object-aligned slices instead of view-aligned slices. The scalar data set is stored in stacks of 2D textures that fill the entire bounding box of the volume. Three stacks of textures are required, one for each of the main coordinate axes. Since 2D slicing is closely related to 3D slicing, only the differences between both approaches are mentioned here.

Data traversal is based on rendering slice polygons. Here, however, the polygons are parallel to one of the faces of the cube-shaped volume data set. Figure 2.2 illustrates the layout of those axis-aligned slices. One of the three stacks of 2D textures is chosen as input for the rendering process. The stack direction is determined by minimizing the angle between viewing direction and processing axis. Similarly to data traversal, the interpolation stage is affected by changing from 3D to 2D slicing. Only bilinear interpolation is applied as the object-aligned slice polygons have a one-to-one correspondence to respective 2D textures of the data set. Due to more coherent memory access and fewer numerical operations, bilinear interpolation is faster than trilinear interpolation, i.e., 2D slicing is usually faster than 3D slicing. However, trilinear interpolation is possible on additional slices [340] by applying multi-texturing. Similarly, additional copies for three texture stacks can be avoided at the cost of decreased rendering performance [340]. All other stages (gradient computation, classification, shading, compositing) are analogous to 3D slicing.

In conclusion, 3D texture slicing offers some advantages compared to the alternative of using stacks of 2D textures. First, only one third of the texture memory is required for the 3D texture method; the 3D approach just holds a single instance of the volume, whereas the 2D approach has to store the complete volume for each of the three main axes. Second, view-aligned slicing through a 3D texture avoids the artifacts that occur when texture stacks are switched. Third, the intrinsic trilinear interpolation in 3D textures directly

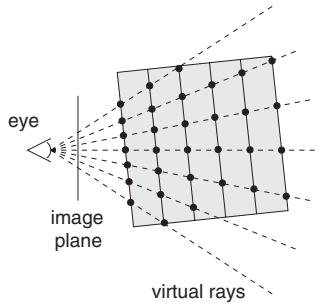


Fig. 2.2. 2D texture slicing. Axis-aligned slices are rendered, leading to sampling positions marked by dots. The dashed rays indicate onto which pixels the volume samples are projected

allows for an arbitrary number of slices with an appropriate resampling on these slices, i.e., the quality of the rendering can be easily adjusted by adapting the slice distance. The main advantage of 2D slicing is its support by almost any graphics hardware and its higher rendering performance.

Most of the rendering techniques in later parts of this book are compatible with 2D and 3D slicing alike. All other cases are specifically noted.

2.3.3 Ray Casting

Ray casting is an image-order approach for volume rendering. The starting point is the image plane, which is scanned in a pixel-by-pixel fashion. Typically, a viewing ray is traversed from the eye point through each pixel into the volume. For improved image quality, supersampling on the image plane leads to more than a single ray per pixel. While a ray is being traversed, the volume rendering integral is evaluated for that ray. The natural compositing order is front-to-back, i.e., from the eye into the volume. Figure 2.3 illustrates the principle of ray casting. Ray casting can be regarded as the volume-rendering analogue of traditional ray tracing for surface-based geometry according to Whitted [489]. One important difference is that ray casting does not spawn secondary rays like ray tracing does.

Ray casting is often used for data given on uniform or Cartesian grids. Here, it is common practice to employ an equidistant sampling along the viewing rays, as shown in Fig. 2.3. Equidistant sampling leads to the Riemann sum approximation (2.4) of the volume rendering integral. An interpolation filter is applied during sampling because the sampling positions usually differ from grid locations. Trilinear interpolation is the most popular reconstruction filter for uniform grids.

The ray casting algorithm can be described by the pseudo code from Fig. 2.4. Data traversal is split in two parts. In the first part – ray setup – the initial sample position along the ray is determined by computing the

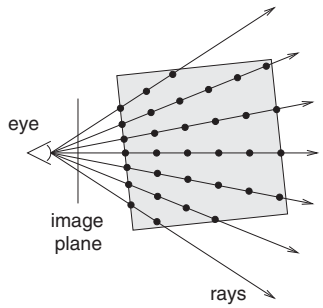


Fig. 2.3. Ray casting idea. Rays are traversed from the eye into the volume. The volume is sampled at positions along the rays (indicated by dots) to compute the volume rendering integral