DISSERTATION

High-Quality Visualization and Filtering of Textures and Segmented Volume Data on Consumer Graphics Hardware

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller,
Institut E186 für Computergraphik und Algorithmen,

und

Dipl.-Ing. Dr.techn. Helwig Hauser,
VRVis Zentrum für Virtual Reality und Visualisierung,

eingereicht an der Technischen Universität Wien,
Fakultät für Informatik,

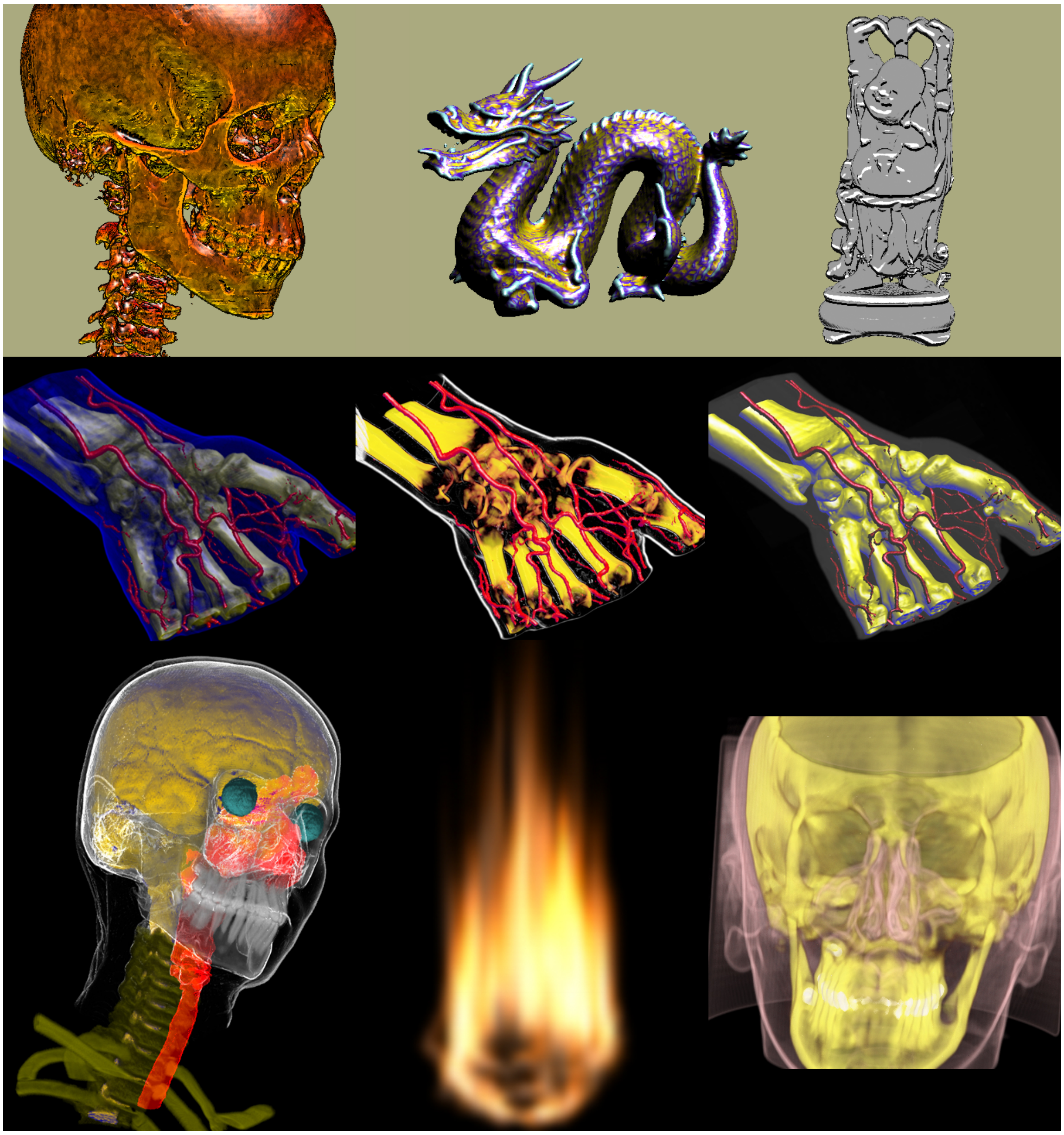
von

Dipl.-Ing. Markus Hadwiger,
Matrikelnummer 9425555,
Laudongasse 34/2/10,
A-1080 Wien

Wien, im Mai 2004

High-Quality Visualization and Filtering of Textures and Segmented Volume Data on Consumer Graphics Hardware

Markus Hadwiger



mailto:msh@VRVis.at
http://www.VRVis.at/vis/resources/diss-MH/

This thesis is dedicated to
the memory of Christoph Berger,
and to my parents,
Dr. Alois and Ingrid Hadwiger.

Abstract

Most rendering methods in visualization and computer graphics are focusing either on image quality in order to produce "correct" images with non-interactive rendering times, or sacrifice quality in order to attain interactive or even real-time performance. However, the current evolution of graphics hardware increasingly allows to combine the quality of off-line rendering approaches with highly interactive performance. In order to do so, new and customized algorithms have to be developed that take the specific structure of graphics hardware architectures into account.

The central theme of this thesis is combining high rendering quality with real-time performance in the visualization of sampled volume data given on regular three-dimensional grids. More generally, a large part of this work is concerned with high-quality filtering of texture maps, regardless of their dimension. Harnessing the computational power of consumer graphics hardware available in off-the-shelf personal computers, algorithms that attain a level of quality previously only possible in off-line rendering are introduced.

A fundamental operation in visualization and computer graphics is the reconstruction of a continuous function from a sampled representation via filtering. This thesis presents a method for using *completely arbitrary convolution filters* for *high-quality reconstruction exploiting graphics hardware*, focusing on real-time magnification of textures during rendering. High-quality filtering in combination with MIP-mapping is also illustrated in order to deal with texture minification. Since texturing is a very fundamental operation in computer graphics and visualization, the resulting quality improvements have a wide variety of applications, including static texture-mapped objects, animated textures, and texture-based volume rendering. The combination of high-quality filtering and all major approaches to hardware-accelerated volume rendering is demonstrated.

In the context of volume rendering, this thesis introduces a framework for *high-quality rendering of segmented volume data*, i.e., data with object membership information such as segmented medical data sets. High-quality shading with per-object optical properties such as rendering modes and transfer functions is made possible, while maintaining real-time performance. The presented method is able to filter the boundaries between different objects on-the-fly, which is non-trivial when more than two objects are present, but important for high-quality rendering.

Finally, several approaches to *high-quality non-photorealistic volume rendering* are introduced, a concept that is especially powerful in combination with segmented volume data in order to focus a viewer's attention and separate focus from context regions. High-quality renderings of isosurfaces are obtained from volumetric representations, utilizing the concept of deferred shading and deferred computation of high-quality differential implicit surface properties. These properties include the gradient, the Hessian matrix, and principal curvature magnitudes as well as directions. They allow high-quality shading and a variety of non-photorealistic effects building on implicit surface curvature.

We conclude that it is possible to bridge the gap between traditional high-quality offline rendering and real-time performance without necessarily sacrificing quality. In an area such as volume rendering that can be very demanding with respect to quality, e.g., in medical imaging, but whose usefulness increases significantly with higher interactivity, combining both high quality and high performance is especially important.

Kurzfassung

Die meisten Renderingmethoden in der Visualisierung und Computergraphik konzentrieren sich entweder auf die Bildqualität, und generieren “korrekte” Bilder mit nicht mehr interaktiven Bildraten, oder opfern die Darstellungsqualität, um interaktive Performance zu erreichen. Andererseits erlaubt es die momentane Entwicklung im Bereich der Graphikhardware zunehmend, die Qualität von Offline Rendering-Ansätzen mit interaktiver Performance zu kombinieren. Um dies auch tatsächlich nutzen zu können, müssen neue und angepasste Algorithmen entwickelt werden, die die spezielle Architektur von Graphikhardware berücksichtigen.

Das zentrale Thema dieser Arbeit ist, hohe Renderingqualität mit Echtzeitfähigkeit bei der Visualisierung von diskreten Volumendaten auf regulären dreidimensionalen Gittern zu kombinieren. Ein wesentlicher Teil beschäftigt sich mit dem generellen Filtern von Texturen unabhängig von deren Dimension. Mit Hilfe der Leistungsfähigkeit heutiger PC Graphikhardware werden Algorithmen demonstriert, die einen Qualitätsstandard erreichen, der bislang nur im Offline Rendering möglich war.

Eine grundlegende Operation in der Visualisierung und Computergraphik ist die Rekonstruktion einer kontinuierlichen Funktion aus einer diskreten Darstellung mittels Filterung. Diese Arbeit stellt eine Methode zur Filterung mit Hilfe von Graphikhardware vor, die prinzipiell beliebige Faltungskerne auswerten kann. Die Hauptanwendung ist hierbei die Vergrösserung von Texturen direkt während dem Rendern. Darüber hinaus kann sie aber auch mit MIP-mapping zur Texturverkleinerung kombiniert werden.

Im Bereich der Volumenvisualisierung stellt diese Arbeit weiters einen Ansatz zur Echtzeitdarstellung von segmentierten Daten vor. Segmentierte Volumendaten haben speziell in medizinischen Anwendungen hohe Bedeutung.

Darüber hinaus stellt diese Arbeit Ansätze zum nicht-photorealistischen Rendern mit hoher Qualität vor, die sich besonders gut eignen, um die Aufmerksamkeit des Betrachters auf bestimmte Fokusbereiche zu lenken. Weiters werden Isoflächen mit Hilfe eines Deferred-Shading Ansatzes dargestellt, wobei differentialgeometrische Eigenschaften, wie beispielsweise die Krümmung der Oberfläche, in Echtzeit berechnet und für eine Vielzahl von Effekten verwendet werden können.

Wir schliessen aus den erreichten Resultaten, dass es möglich ist, die Lücke zwischen Offline Rendering mit hoher Qualität auf der einen Seite, und Echtzeitrendering auf der anderen Seite, zu schliessen, ohne dabei notwendigerweise die Qualität zu beeinträchtigen. Besonders wichtig ist dies im Bereich des Renderings von Volumendaten, das sehr oft hohe Qualitätsansprüche hat, etwa bei der Darstellung von medizinischen Daten.

Contents

Abstract, Kurzfassung **v**

Related Publications **ix**

1 Introduction and Overview **1**
1.1 The GPU – Real-Time High-Quality Rendering and More 1
1.2 Contribution 2
1.3 Organization 4

2 Fundamentals and State of the Art **7**
2.1 Consumer Graphics Hardware 7
2.2 Sampling and Reconstruction 15
2.3 Texture Filtering 17
2.4 Volume Rendering 20
2.5 Texture-Based Volume Rendering 30

3 High-Quality Filtering **39**
3.1 Magnification Filters 41
3.2 Minification Filters and MIP-Mapping 54
3.3 Surface and Solid Texture Filtering 60
3.4 Volume Texture Filtering 65
3.5 Error and Quality Considerations 69

4 Volume Rendering of Segmented Data **81**
4.1 Segmented Data Representation 83
4.2 Rendering Segmented Data 84
4.3 Boundary Filtering 90
4.4 Two-Level Volume Rendering 93
4.5 Performance 97

5 Non-Photorealistic Volume Rendering **101**
5.1 Basic Non-Photorealistic Rendering Modes 101
5.2 Isosurfaces 104
5.3 Deferred Shading 105
5.4 Deferred Gradient Reconstruction 107
5.5 Other Differential Properties 108
5.6 Rendering from Implicit Curvature 111

6 The HVR Volume Rendering Framework 117
6.1 Basic Structure 118
6.2 Hardware Independence 122
6.3 Shaders 124
6.4 User Interface 125

7 Summary 131
7.1 High-Quality Filtering 131
7.2 Volume Rendering of Segmented Data 134
7.3 Non-Photorealistic Volume Rendering 136

8 Conclusions 139

A Meta MIP-Map Details 141

B OpenGL Extensions 145

Acknowledgments 147

Curriculum Vitae 149

Bibliography 151

Related Publications

This thesis is based on the following publications:

Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller,
Hardware-Accelerated High-Quality Filtering on PC Hardware,
Proceedings of Vision, Modeling, and Visualization (VMV) 2001, 2001, pp. 105-112.

Markus Hadwiger, Ivan Viola, Thomas Theußl, and Helwig Hauser,
Fast and Flexible High-Quality Texture Filtering with Tiled High-Resolution Filters,
Proceedings of Vision, Modeling, and Visualization (VMV) 2002, 2002, pp. 155-162.

Markus Hadwiger, Helwig Hauser, and Torsten Möller,
Quality Issues of Hardware-Accelerated High-Quality Filtering on PC Graphics Hardware,
Proceedings of the 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) 2003, 2003, pp. 213-220.

Markus Hadwiger, Christoph Berger, and Helwig Hauser,
High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware,
Proceedings of IEEE Visualization 2003, 2003, pp. 301-308.

and the following technical report:

Christian Sigg, Markus Hadwiger, Markus Gross, and Katja Bühler,
Real-Time High-Quality Rendering of Isosurfaces,
TR-VRVis-2004-015, 2004, VRVis Research Center.

This thesis is also related to the following technical sketches presented at the annual SIGGRAPH conference in the U.S.:

Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller,
Hardware-Accelerated High-Quality Filtering of Solid Textures,
SIGGRAPH 2001 Conference Abstracts and Applications, 2001, p. 194.

Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller,
MIP-Mapping with Procedural and Texture-Based Magnification,
SIGGRAPH 2003 Sketches and Applications, 2003.

Chapter 1

Introduction and Overview

This chapter gives an overview of the background and motivation of this thesis, as well as of its contribution to the current state of the art. It concludes with an overview of its organization.

The major motivation for this thesis is to combine high rendering quality with interactivity, especially in the context of volume rendering. Volume data have several important applications, including medical imaging of CT (computed tomography) or MRI (magnetic resonance imaging) scans, and numerical simulations of gaseous phenomena or participating media. However, although most commonly viewed as being comprised of a cloud of particles of a certain density, it is also important to bear in mind that the original object that has been scanned in order to obtain volume data often consists of clearly distinct objects with definite bounding surfaces, e.g., a scan of the human body, instead of semi-transparent structures. Additionally, volumetric representations are an important approach to modifying or deforming object surfaces given in implicit form, where the desired surface can be extracted for rendering as an isosurface.

The major vehicle to allow the desired combination of high rendering quality and interactive performance is the computational power of consumer graphics hardware, especially the recent development of programmable graphics hardware, which is now often called the GPU.

1.1 The GPU – Real-Time High-Quality Rendering and More

The huge demand for high-performance 3D computer graphics generated by computer games has led to the availability of extremely powerful 3D graphics accelerators in the consumer marketplace. These graphics cards by now not only rival, but in almost all areas even surpass, the tremendously expensive graphics workstations from just a couple of years ago. Current state of the art consumer graphics chips such as the NVIDIA GeForce FX [117], or the ATI Radeon 9800 [1], offer a level of programmability and performance that not only makes it possible to perform traditional workstation tasks on a cheap personal computer, but even enables the use of rendering algorithms that previously could not be employed in real-time graphics at all.

Probably even more importantly, the recent integration of highly programmable vertex and pixel shading units [136], including the availability of high-level shading languages [136] and floating point precision and range for computations, has ushered in a shift toward thinking of these graphics chips as *GPUs*, or *graphics processing units*, that are almost general stream processors [64], instead of mere graphics accelerators, in analogy to the main CPU. And in fact, recent developments in GPU research are using these processors for much more than graphics [30], including general computations such as solving large linear systems [9, 79], non-linear optimization [49], simulation [41], segmentation of medical images and volumes [86], and other applications of numerical computing. This allows to combine real-time computation of data with real-time rendering for visualizing them, and further mandates a move toward higher quality standards than previously thought sufficient for real-time rendering.

Volume rendering and GPUs

Traditionally, volume rendering has especially high computational demands due to the enormous amount of data that needs to be processed. One of the major problems of using consumer graphics hardware for volume rendering is the amount of texture memory required to store the volume data, and the corresponding bandwidth consumption when texture fetch operations cause basically all of these data to be transferred over the bus for each rendered frame.

However, the increased performance, on-board memory, bus bandwidth, and especially the programmability of consumer graphics hardware today allows real-time high-quality volume rendering, for instance with respect to the application of transfer functions [70], shading [72], and filtering [144], both for regular grids [24], and unstructured grids [166]. In spite of the tremendous requirements imposed by the sheer amount of data contained in a volume, the flexibility and quality, but also the performance, that can be achieved by volume renderers for consumer graphics hardware is increasing rapidly, and has made possible entirely new algorithms for high-quality volume rendering.

1.2 Contribution

The goal of this thesis is to advance the state of the art in real-time high-quality volume rendering and texture filtering on consumer graphics hardware.

This thesis introduces a general framework for high-quality filtering of texture maps with arbitrary convolution filters. In practice, cubic convolution filters are a very attractive alternative to the standard linear interpolation supported natively by graphics hardware.

This thesis presents an efficient method for rendering segmented volume data, such as segmented medical data sets from CT or MRI modalities, with per-object rendering modes, transfer functions, and even per-object compositing modes utilizing a minimal number of rendering passes. Using per-object compositing modes and a single global compositing mode combining the contributions of individual objects is known as *two-level volume rendering* [43, 44]. High rendering quality is achieved by evaluating shading equations on a per-pixel basis and filtering object boundaries with pixel resolution even when more than two objects are contained in the segmented volume.

The power of combining traditional volume rendering techniques and non-photorealistic methods in a single image on a per-object basis is illustrated. The use of high-quality filtering with cubic filters can be used very effectively for the deferred shading and deferred computation of differential properties of isosurfaces. Deferred computations allow to compute high-quality principal curvature information via tri-cubic convolution in real-time, which is a powerful basis for non-photorealistic rendering based on implicit surface curvature.

In this thesis, we restrict ourselves to volume data defined on *regular* or *Cartesian* grids, which is the most common type for volume data, especially in medical applications. Due to the regular structure of texture maps, it is also the grid type most conducive to hardware rendering. In such grids, the volume data are comprised of samples located at grid points that are equispaced along each respective volume axis, and can therefore easily be stored in a texture map. However, unstructured grids can be handled by re-sampling them onto a regular grid before rendering [88, 165, 176].

Texture filtering

A fundamental operation that is crucial for the resulting quality in volume rendering is the reconstruction of the original continuous volume from the sampled representation via filtering. In the context of texture-based volume rendering on consumer graphics hardware, where the volume data are stored in texture maps, this reconstruction is achieved via *texture filtering*.

However, on current consumer graphics hardware, texture maps are most commonly filtered and re-sampled during rendering with a combination of nearest-neighbor interpolation, linear interpolation, and MIP-mapping [178]. For example, in OpenGL, the basic texture filtering mode specifying this combination is part of the texture object itself and is used for all accesses to the corresponding texture map [106].

Linear interpolation for texture filtering can lead to highly visible artifacts, especially when a texture is magnified significantly, e.g., when being viewed up close. If higher filtering quality is desired, filters of higher order (than linear) have to be used. A good trade-off between speed and quality are cubic filters [67]. Although a lot of research has been done to investigate high-quality reconstruction filters, they are usually considered to be much too computationally expensive for use in interactive applications [102].

We demonstrate that using higher-order convolution filters for high-quality texture magnification in real-time is possible on current graphics hardware and introduce a framework with a family of algorithms offering different trade-offs between quality, speed, texture memory usage, and flexibility. The basic approach of our framework builds on filter kernels stored entirely in texture maps and requires only basic multi-texturing capabilities. Moreover, the programmability of more recent hardware can be used for many optimizations, yielding single-pass cubic filtering of textures, for example, as also illustrated in this thesis.

Although in applications such as volume rendering, textures are commonly used without MIP-mapping (because they are magnified most of the time), many other applications have to deal with both magnification and minification of textures. This thesis also demonstrates how higher-order convolution filters can be combined with MIP-mapping in order to deal with minification of textures in addition to magnification. This enables their use in a wide variety of applications.

Volume rendering of segmented data

In many volume rendering methods, all voxels contained in a volumetric data set are treated in an identical manner, i.e., without using any a priori information that specifies object membership on a per-voxel basis.

In that case, visual distinction of objects is usually achieved by either using multiple semi-transparent iso-surfaces [58], or with direct volume rendering and an appropriate transfer function [69]. In the latter case, multi-dimensional transfer functions have proven to be especially powerful in facilitating the perception of different objects [66, 69, 70].

In recent years, non-photorealistic volume rendering approaches have also been used successfully for improving the perception of distinct objects embedded in a single volume [98, 135].

However, it is also often the case that a single rendering method or transfer function does not suffice in order to distinguish multiple objects of interest according to a user's specific needs. A very powerful approach to tackling this problem is to create explicit object membership information via segmentation [159], which usually yields a binary segmentation mask for objects of interest, or an object ID for each of the volume's voxels.

Real-time high-quality volume rendering of segmented data on consumer graphics hardware is not a trivial extension of standard volume rendering approaches. This thesis, however, introduces a framework that allows to do so. We demonstrate the use of flexible optical properties specified on a per-object basis, including per-object transfer functions, a variety of rendering modes each of which can be assigned to any object, and per-object compositing modes in combination with a single global compositing mode, i.e., two-level volume rendering [43, 44]. We also present a method for filtering the boundaries of objects in order to achieve high-quality separation of objects.

Non-photorealistic volume rendering

Non-photorealistic techniques, e.g., methods imitating the style of technical illustrations [28], are very powerful in conveying a specific meaning, attracting the viewer's attention, or simply providing context to a focus region of interest rendered with a more traditional style, and are increasingly being applied to volume rendering [16, 67, 99, 114, 135, 155].

We demonstrate high-quality non-photorealistic volume rendering in the context of rendering segmented volume data. As examples we show tone shading [28] adapted to volume rendering, and a simple contour rendering model [16]. The corresponding shading equations are computed on-the-fly on a per-fragment basis, which yields high-quality results.

Additionally, we present a deferred shading pipeline for rendering of isosurfaces with high-quality gradients and additional differential surface properties. Isosurfaces of volumetric data are an important part of volume visualization, and we apply a series of tri-cubic filtering passes in a deferred shading stage in order to compute isosurface gradients, additional differential properties such as the Hessian matrix and principal curvature magnitudes and directions [67], and finally perform shading from these quantities. Only the first stage of intersecting viewing rays with the isosurface operates in object space, while all surface properties and shading are exclusively computed in screen space. Thus, most performance-critical tasks are proportional to the resolution of the two-dimensional output instead of the three-dimensional volume

High-quality curvature information can be used for a variety of non-photorealistic surface shading effects, and we demonstrate contour rendering with constant screen space thickness controlled by curvature, color mapping curvature magnitudes and derived quantities such as mean or Gaussian curvature, and advecting flow in the directions of principal curvature. Visualizing curvature information can also provide the basis for surface investigation tasks.

1.3 Organization

Chapter 2 presents fundamentals of consumer graphics hardware architecture, sampling and reconstruction, hardware texture mapping and filtering, and volume rendering. The introduction to volume rendering focuses on texture-based volume rendering, which is very important in the context of this thesis. Chapter 2 also reviews the state of the art of high-quality filtering in general, and volume rendering on consumer graphics hardware in particular.

The following three chapters present the main contributions of this thesis in high-quality filtering (chapter 3), volume rendering of segmented data (chapter 4), and non-photorealistic volume rendering (chapter 5).

The thesis concludes with a summary of the main contributions, conclusions and implications of this work, as well as an extensive bibliography.

Chapter 2

Fundamentals and State of the Art

This chapter reviews essential fundamentals of consumer graphics hardware architecture, sampling and reconstruction of signals and textures, hardware texture mapping, volume rendering, and texture-based volume rendering. It also includes an overview of the current state of the art and prior work related to this thesis in the two major areas of filtering and reconstruction, as well as volume rendering with a focus on exploiting graphics hardware with texture-mapping capabilities for interactive performance.

Parts of this chapter are based on the course *High-Quality Volume Graphics on Consumer PC Hardware* presented at SIGGRAPH 2002, and the related course notes [23].

2.1 Consumer Graphics Hardware

We first give a brief overview of the operation of graphics hardware in general, and continue with a description of the kind of graphics hardware that is most interesting in the context of this thesis, i.e., consumer graphics hardware such as the NVIDIA GeForce family [117], and the ATI Radeon series [1]. We give an overview of the evolution and capabilities of these architectures, especially focusing on per-fragment programmability. The application programming interface (API) used in this and all other parts of this thesis is OpenGL [142]. Basic OpenGL extensions that are useful for texture-based volume rendering are described in Appendix B. The most recent developments have led to the availability of high-level shading languages for programmable graphics hardware (GPUs), e.g., the OpenGL shading language [136] or Cg [27], substituting earlier OpenGL extensions and assembly-level shaders.

The Graphics Pipeline

On current consumer graphics hardware, all geometry consists of a set of triangles, lines, and points, which are ultimately turned into pixels via the process of *rasterization*. The sequence of operations that turns a geometric scene description into a raster image is usually depicted in the form of a pipeline of sequential operations, i.e., the *graphics pipeline* illustrated in figure 2.1. The basic input to the graphics pipeline is a stream of vertices. Primitives such as triangles can be formed from individual vertices via connectivity information. The output of the pipeline is a raster image that can be displayed on the screen.

Until two years ago, practically all graphics hardware used a more or less identical pipeline structure of fixed-function stages. However, the most recent GPU architectures have intro-

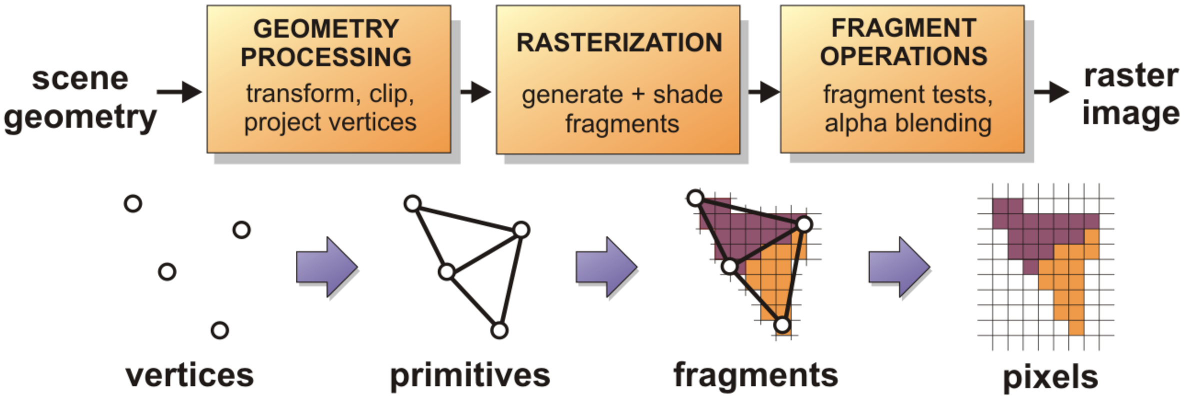


Figure 2.1: The graphics pipeline turns a geometric scene description into the pixels of a raster image through a sequence of operations [23].

duced a large amount of programmability into the basic graphics pipeline, especially with respect to operations on vertices and fragments [136]. Still, the graphics pipeline can roughly be divided into three different stages [106]:

Geometry processing. The first stage of the graphics pipeline operates on vertex information. Standard operations include affine transformations in order to rotate, translate, or scale geometry, and the computation of per-vertex lighting. Through their vertices, the geometric primitives themselves are transformed along implicitly. In a programmable pipeline, these vertex operations can be specified entirely by a user-written program, which is called a *vertex program* or *vertex shader* [136]. After per-vertex computations, they are connected to form primitives, clipped to the view frustum, and projected to screen space where they will be rasterized subsequently.

Rasterization. The next stage decomposes already projected primitives into *fragments*. A fragment is closely related to a pixel in the final image, but it may be discarded by one of several tests that follow rasterization. After a fragment has initially been generated by the rasterizer, colors fetched from texture maps are applied, followed by further color operations, often subsumed under the term *fragment shading*. On today's programmable consumer graphics hardware, both fetching colors from textures and additional color operations applied to a fragment are programmable to a large extent, and specified in a *fragment program* or *fragment shader* [136].

Fragment operations. After fragments have been generated and shaded, several tests are applied that finally decide whether an incoming fragment is discarded or displayed on the screen as a pixel. These tests include alpha testing, stencil testing, and depth testing. After fragment tests have been applied and the fragment has not been discarded, it is combined with the previous contents of the frame buffer, a process known as *alpha blending* [106]. After this, the fragment has become a pixel.

In order to understand algorithms exploiting graphics hardware, especially GPUs, the order of operations in the graphics pipeline is crucial. In the following sections, we describe each of the major three stages outlined above in more detail.

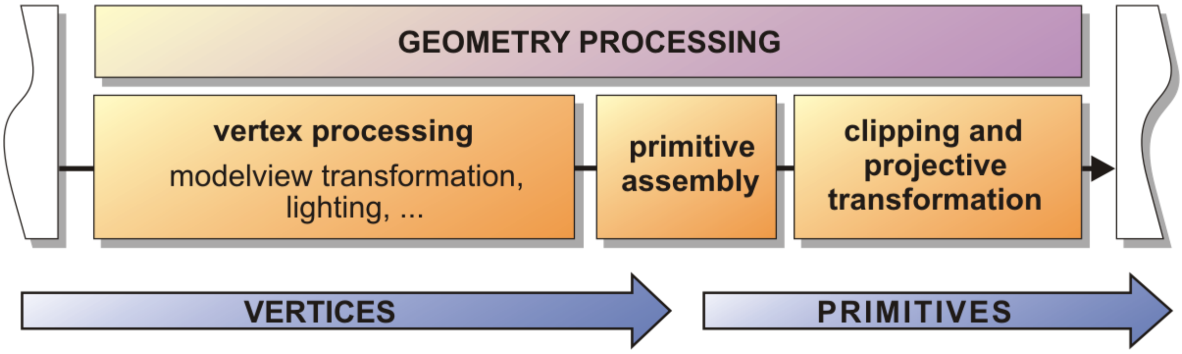

GEOMETRY PROCESSING
vertex processing
modelview transformation, lighting, ...
primitive assembly
clipping and projective transformation
VERTICES
PRIMITIVES

Figure 2.2: Geometry processing as part of the graphics pipeline [23].

Geometry Processing

On recent GPU architectures, the geometry processing stage consists of a highly programmable *vertex processing unit* or *vertex processor* [136], and fixed-function units for assembling primitives from a stream of vertices, and clipping and projecting the resulting geometry. These stages are illustrated in figure 2.2. On programmable architectures, vertex processing can be specified via a user-supplied assembly language program or a shader written in a high-level shading language [136]. Vertex processing is performed on a per-vertex basis following a stream processing model [64]. The major tasks usually performed by a *vertex shader* in the vertex processor are:

Modelview transformation. The transformation from object space, where vertex coordinates are originally specified, to view space is specified in a single 4 × 4 matrix using homogeneous coordinates. This matrix subsumes both the transformation from object space to world space, placing objects in relation to one another, and the transformation from world space to view space, transforming everything into the coordinate system of the camera [106].

Lighting. Many lighting models are evaluated on a per-vertex instead of a per-fragment basis. Moreover, although more complex lighting computations are increasingly carried out in the fragment shader, the vertex shader usually still performs several setup computations needed for the subsequent per-fragment shading.

The programmable vertex processing is followed by the following fixed-function operations:

Primitive assembly. Since per-vertex processing operates on an unconnected stream of vertices, actual primitives such as triangles must be generated by assembling them from the vertex data in the stream.

Clipping. Primitives must be clipped against the view frustum in order to prevent unnecessary processing of invisible fragments, i.e., fragments outside the output image, in the subsequent rasterization stage.

Projective transformation. Multiplication with a projective 4 × 4 matrix computes the projection of primitives onto the plane of the output image.

After the final stage of geometry processing, all operations are performed in two-dimensional screen space, i.e., the plane of the output image.

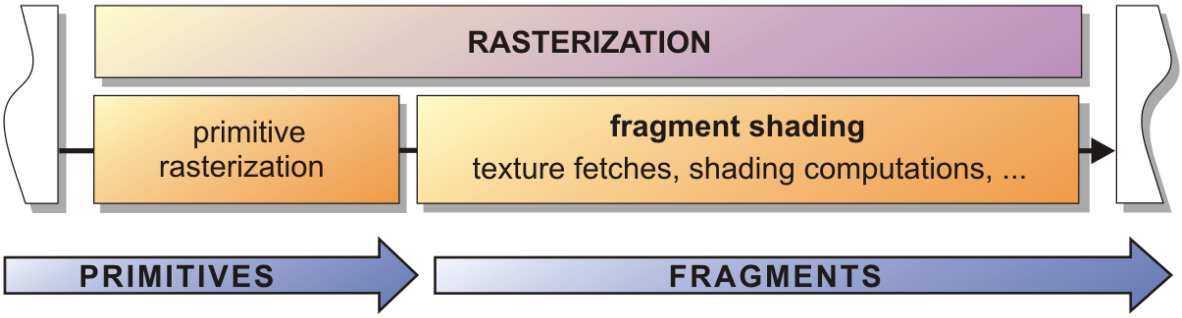


Figure 2.3: Rasterization as part of the graphics pipeline [23].

Rasterization

The rasterization stage turns geometric primitives into a *stream of fragments* corresponding to pixels in the output image. It is illustrated in figure 2.3. The actual primitive rasterization generates fragments and interpolates per-vertex attributes such as colors and texture coordinates over the interior of primitives in order to generate interpolated per-fragment attributes. This fixed-function stage is followed by a programmable fragment shading stage. On current GPU architectures, the *fragment processor* [136] executes a *fragment program* or *fragment shader* for highly flexible shading computations. The major tasks usually performed by a fragment shader are:

Texture fetch. Textures are mapped onto polygons according to texture coordinates specified at the vertices. For each fragment, these texture coordinates must be interpolated and a texture lookup is performed at the resulting coordinate. This process yields an interpolated color value *fetched* from the texture map. In today's consumer graphics hardware from two to sixteen textures can be fetched simultaneously for a single fragment. Furthermore, the lookup process itself can be controlled easily, for example by routing colors back into texture coordinates, which is known as *dependent texturing*.

Fragment shading. In addition to sampling textures, further color operations are applied in order to shade a fragment. A trivial example would be the combination of texture color and primary, i.e., diffuse, color by simple multiplication.

Today's consumer graphics hardware allows highly flexible control of the entire fragment shading process. Two years ago, the texture fetch and fragment shading stages were completely separate, but on current architectures the texture fetch stage has become an integral part of the fragment shading stage.

Fragment Operations

After a fragment has been shaded, but before it is turned into an actual pixel that is stored in the frame buffer and ultimately displayed on the screen, several *fragment operations* are performed. Most of these operations are *fragment tests* whose outcome determines whether a fragment is discarded, e.g., because it is occluded, or actually becomes a pixel in the output image. The last fragment operation performs *alpha blending* and computes the actual color of the output pixel corresponding to a fragment. The sequence of fragment operations is illustrated in figure 2.4:

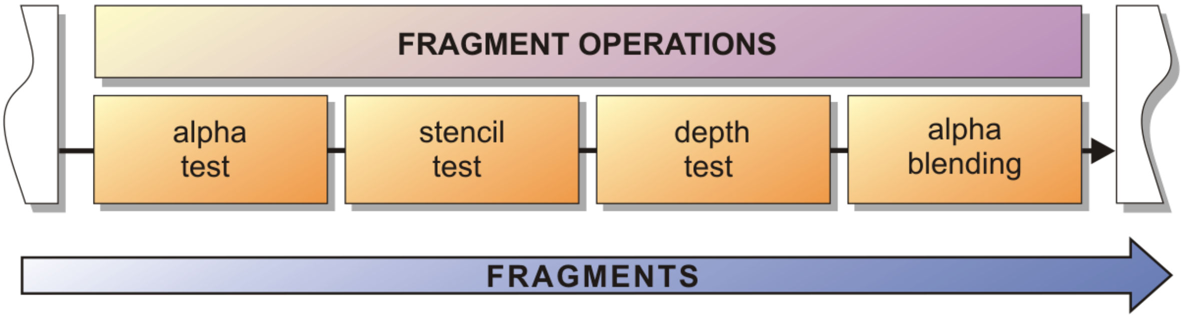


Figure 2.4: Fragment operations as part of the graphics pipeline [23].

Alpha test. A simple comparison of the alpha value of a fragment with a global reference value can be used to selectively discard fragments. In texture-based volume rendering, the alpha test is a common method for rendering isosurfaces [174].

Stencil test. The *stencil buffer* contains a stencil value for each pixel, and a configurable stencil test operation can be used to determine how to update the stencil buffer and whether a fragment should be discarded or not depending on a comparison operation. The stencil test can also take the result of the depth test into account.

Depth test. The standard approach for visibility determination in image space is to use a *Z buffer*, which is also called a *depth buffer*. By comparing a fragment's depth value with the previous depth value at the corresponding pixel location occluded fragments can be discarded easily.

Alpha blending. The last fragment operation evaluates a blend equation for alpha blending [106], e.g., in order to implement the *over operator* for rendering semi-transparent geometry [130]. In texture-based volume rendering, alpha blending is a crucial part of evaluating the volume rendering integral [23]. See also section 2.4.

After the last fragment operation, i.e., alpha blending, has been performed, the output color buffer contains the actual image that will be displayed on the screen.

The basic graphics pipeline illustrated in this section has undergone significant modifications in the last two to three years. Starting with a fixed-function pipeline, more and more programmability has been introduced until current architectures have arrived at a stage where both the vertex and the fragment processor are able to execute very general code that is also not limited to shading computations. The introduction of high-level shading languages [27, 131, 136] in conjunction with more powerful hardware now allows a wide variety of effects and computations. The following section gives an overview of the evolution and capabilities of the consumer graphics hardware architectures that are most important today.

Standard Hardware Platforms

In this section, we briefly discuss the consumer graphics hardware architectures that we are using for high-quality filtering and volume rendering in the context of this thesis. The following sections discuss important features of these architectures in detail. The two currently most important vendors of programmable consumer graphics hardware are NVIDIA [117] and

ATI [1]. The current state of the art consumer graphics chips are the NVIDIA GeForce FX series, and the ATI Radeon 9500+ series.

NVIDIA

In late 1999, the *GeForce 256* introduced hardware-accelerated geometry processing to the consumer marketplace. Before this, transformation and projection was either done by the OpenGL driver on the CPU, or even by the application itself. The first GeForce also offered a configurable mechanism for *fragment shading*, i.e., the register combiners OpenGL extension (`GL_NV_register_combiners`). The focus on programmable fragment shading was even more pronounced during introduction of the *GeForce 2* in early 2000, although it brought no major architectural changes from a programmer's point of view. On the first two GeForce architectures it was possible to use two textures simultaneously in a single pass (*multi-texturing*). Usual boards had 32MB of on-board RAM, although GeForce 2 configurations with 64MB were also available.

The next major architectural step came with the introduction of the *GeForce 3* in early 2001. Moving away from a fixed-function pipeline for geometry processing, the GeForce 3 introduced *vertex programs*, which allowed the programmer to write custom assembly language code operating on vertices. The number of simultaneous textures was increased to four, the register combiners capabilities were improved (`GL_NV_register_combiners2`), and the introduction of texture shaders (`GL_NV_texture_shader`) introduced *dependent texturing* on a consumer graphics platform for the first time. Additionally, the GeForce 3 also supported 3D textures (`GL_NV_texture_shader2`) in hardware. Usual GeForce 3 configurations had 64MB of on-board RAM, although boards with 128MB were also available.

The *GeForce 4*, introduced in early 2002, extended the modes for dependent texturing (`GL_NV_texture_shader3`), offered point sprites, hardware occlusion culling support, and flexible support for rendering directly into a texture (the latter became also possible on a GeForce 3 with the OpenGL drivers released at the time of the GeForce 4). The standard amount of on-board RAM of GeForce 4 boards was 128MB, which was also the maximum amount supported by the chip itself.

The probably most important technological leap in the GeForce series was introduced with the GeForce FX in early 2003, which marks the final transition from mere graphics accelerators to *GPUs* (*graphics processing units*). Vertex and fragment operations became highly programmable, including shading computations in 16-bit and 32-bit floating point precision and range. These features ushered in the era of *general purpose computations on GPUs* [30], where general, not necessarily graphics-related, computations can be performed in a highly parallel manner. On-board memory configurations of 256MB became the new standard.

The current state of the art NVIDIA GPU is the *GeForce 6 FX* series introduced in April 2004 [117]. One of the major changes with respect to earlier GPUs is the support of data-dependent conditional branching in the fragment shader.

ATI

In mid-2000, the original *Radeon* was the first consumer graphics hardware to support 3D textures natively. For multi-texturing, it was able to use three 2D textures, or one 2D and one 3D texture simultaneously. However, fragment shading capabilities were constrained to a few extensions of the standard OpenGL texture environment. The usual on-board configuration was 32MB of RAM.

The *Radeon 8500*, introduced in mid-2001, was a huge leap ahead of the original Radeon, especially with respect to fragment programmability (`GL_ATI_fragment_shader`), which offered a unified model for texture fetching (including flexible dependent textures), and color combination. This architecture also supported programmable vertex operations (`GL_EXT_vertex_shader`), and six simultaneous textures with full functionality, i.e., even six 3D textures could be used in a single pass. The fragment shading capabilities of the Radeon 8500 were exposed via an assembly-language level interface, and very easy to use. Rendering directly into a texture is also supported. On-board memory of Radeon 8500 boards usually was either 64MB or 128MB.

A minor drawback of Radeon OpenGL drivers (for both architectures) is that paletted textures (`GL_EXT_paletted_texture`, `GL_EXT_shared_texture_palette`) are not supported, which otherwise provide a nice fallback for volume rendering when post-classification via dependent textures is not used, and downloading a full RGBA volume instead of a single-channel volume is not desired due to the memory overhead incurred.

The first real GPU ever was the *Radeon 9700* introduced in mid-2002, which offered very high performance, floating point computations, and highly programmable vertex and fragment shading units supporting the `GL_ARB_vertex_program` and `GL_ARB_fragment_program` OpenGL extensions, respectively. An especially important feature of the Radeon 9500+ series is the *early z-test* (also called *early depth test*), which allows to avoid the execution of expensive fragment shaders for fragments that will be culled in the fragment shader itself.

The current state of the art ATI GPU is the *Radeon X800* introduced in May 2004. A major improvement in comparison to the 9500+ series is a higher maximum number of instructions in the fragment shader.

Fragment Shading

Building on the general discussion presented above, we now provide a more detailed discussion of the fragment shading stage of the graphics pipeline, which of all the pipeline stages is the most important one for building a consumer hardware volume renderer.

Although in earlier architectures *texture fetch* and *fragment shading* were two separate stages, current architectures support texture fetch operations as simply one part of overall fragment shading. In these architectures, a texture fetch is just another way of coloring fragments, in addition to performing other color operations.

The terminology related to fragment shading and the corresponding stages of the graphics pipeline has only begun to change after the introduction of the first highly-configurable graphics hardware architecture, i.e., the original NVIDIA GeForce family. Before this, fragment shading was so simple that no general name for the corresponding operations was used. The traditional OpenGL model assumes a linearly interpolated primary color (the diffuse color) to be fed into the first texture unit, and subsequent units (if at all supported) to take their input from the immediately preceding unit. Optionally, after all the texture units, a second linearly interpolated color (the specular color) can be added in the color sum stage (if supported), followed by application of fog [106]. The shading pipeline just outlined is commonly known as the traditional *OpenGL multi-texturing* pipeline [106].

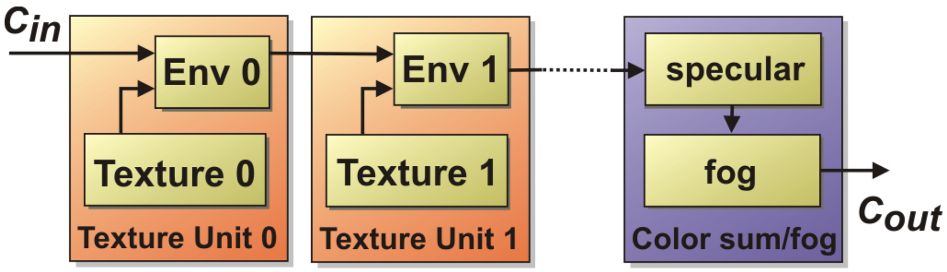


Figure 2.5: The traditional OpenGL multi-texturing pipeline [23]. Conceptually identical texture units (orange) are cascaded up to the number of supported units, followed by fog and specular color application (blue), which generates the final fragment color.

Traditional OpenGL Multi-Texturing

Before the advent of programmable fragment shading (see below), the prevalent model for shading fragments was the traditional OpenGL multi-texturing pipeline [106], which is depicted in figure 2.5. The primary (or diffuse) color, which has been specified at the vertices and linearly interpolated over the interior of a triangle by the rasterizer, is the intial color input to the pipeline. The pipeline itself consists of several texture units, each of which has exactly one external input (the color from the immediately preceding unit, or the initial fragment color in the case of unit zero), and one internal input (the color sampled from the corresponding texture). The *texture environment* of each unit (specified via `glTexEnv*()`) determines how the external and the internal color are combined. If the unit was the last one, a second linearly interpolated color can be added in a *color sum* stage (if `GL_EXT_separate_specular_color` is supported), followed by optional fog application. The output of this cascade of texture units and the color sum and fog stage becomes the shaded fragment color.

Standard OpenGL supports only very simple texture environments, i.e., modes of color combination, such as multiplication and blending. For this reason, several extensions have been introduced that add more powerful operations. For example, dot-product computation via `GL_EXT_texture_env_dot3` (see Appendix B).

Programmable Fragment Shading

Although entirely sufficient only a few years ago, the OpenGL multi-texturing pipeline has a lot of drawbacks, is inflexible, and cannot accommodate the capabilities of today's consumer graphics hardware. Most of all, colors cannot be routed arbitrarily, but are forced to be applied in a fixed order, and the number of available color combinations is very limited. Furthermore, the color combination not only depends on the setting of the corresponding texture environment, but also on the internal format of the texture itself, which prevents using the same texture for different purposes, especially with respect to treating the RGB and alpha channels separately.

For these and other reasons, fragment shading has become programmable in its entirety on the latest graphics hardware architectures. The first OpenGL extension that could be considered to be in the context of fragment shading were the original NVIDIA register combiners. They were comprised of a register-based execution model and programmable input and output routing and color combination operations.

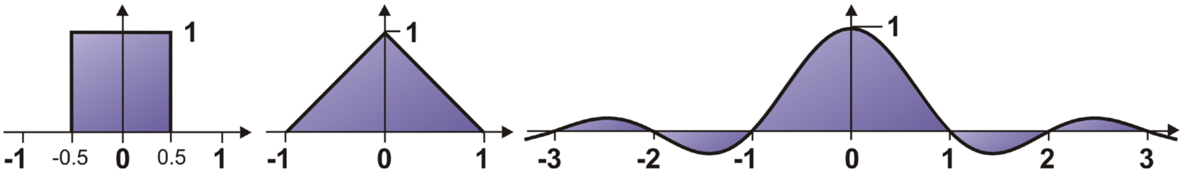


Figure 2.6: Different reconstruction filters: box (left), tent (center), and sinc filter (right).

The current state of the art is writing a *fragment shader* in an assembly language (usually using the `GL_ARB_fragment_program` OpenGL extension), or in a high-level shading language such as the OpenGL shading language [136], or NVIDIA's Cg [27]. These fragment shaders are specified as a string of statements, compiled by the OpenGL driver, and downloaded to the graphics hardware where they are executed for each fragment.

2.2 Sampling and Reconstruction

When continuous functions need to be handled within a computer, a common approach is to convert them to a discrete representation by *sampling* the continuous domain at – usually equispaced – discrete locations [119]. In addition to the discretization that is done with respect to location, the individual samples also have to be quantized in order to map continuous scalars to quantities that can be represented as a discrete number, which is usually stored in either fixed-point, or floating-point format. After the continuous function has been converted into a discrete function via sampling, this function is only defined at the exact sampling locations, but not over the original continuous domain. In order to once again be able to treat the function as being continuous, a process known as *reconstruction* must be performed, i.e., reconstructing a continuous function from a discrete one [119].

Reconstruction is often performed by applying a reconstruction filter to the discrete function, which is usually done by performing a convolution of the filter kernel (the function describing the filter) with the discrete function:

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} f_i \cdot h(x - i) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} f_i \cdot w_i(x) \tag{2.1}$$

where $g(x)$ is the output at re-sampling position x, f_i is the discrete input function, $h(x)$ is the continuous filter kernel, $m = n/2$ is half the filter width when n is the order (cubic: $n = 4$), and the $w_i(x)$ are the n weights corresponding to x.

The simplest such reconstruction filter $h(x)$ is known as the box filter (figure 2.6 left), which results in nearest-neighbor interpolation of the sampled function. Another reconstruction filter that is commonly used, especially in hardware, is the tent filter (figure 2.6 center), which results in linear interpolation.

In general, we know from sampling theory [119] that a continuous function can be reconstructed exactly if certain conditions are honored during the sampling process. The original function must be band-limited, i.e., not contain any frequencies above a certain threshold, and the sampling frequency must be at least twice as high as this threshold (which is often called the Nyquist frequency [119]). The requirement for a band-limited input function is often enforced by applying a low-pass filter before the function is sampled, if this is possible. Low-pass filtering discards frequencies above the Nyquist limit, which would otherwise result in aliasing, i.e., high frequencies being interpreted as much lower frequencies after sampling, due to overlap in the frequency spectrum.

The statement that a function can be reconstructed exactly stays theoretical, however, since, even when disregarding quantization artifacts, the reconstruction filter used would have to be ideal. The "perfect," or ideal, reconstruction filter is known as the *sinc* filter [119], whose frequency spectrum is box-shaped, and described in the spatial domain by the following equation:

$$sinc(x) = \frac{\sin(\pi x)}{\pi x} \tag{2.2}$$

A graph of this function is depicted in figure 2.6 right. The simple reason why the *sinc* filter cannot be implemented in practice is that it has infinite extent, i.e., the filter function is non-zero from $-\infty$ to $+\infty$. Thus, a trade-off between reconstruction time, depending on the extent of the reconstruction filter, and reconstruction quality must be found.

Reconstruction filters

Over the years, a lot of work in computer graphics has been devoted to investigating high-quality reconstruction via convolution, although almost exclusively with software implementations.

Keys [65] derived a family of cardinal splines for reconstruction and showed that among these the Catmull-Rom spline is numerically most accurate. Mitchell and Netravali [108] derived another family of cubic splines quite popular in computer graphics, the BC-splines. Marschner and Lobb [102] compared linear interpolation, cubic splines, and windowed sinc filters. They concluded that linear interpolation is the cheapest option and will likely remain the method of choice for time-critical applications.

Möller et al. provide a general framework for analyzing filters in the spatial domain, using it to analyze the cardinal splines [109], and the BC-splines [110]. They also show how to design accurate and smooth reconstruction filters [111]. Turkowsky [156] used windowed ideal reconstruction filters for image resampling. Theußl et al. [151] used the framework developed by Möller et al. [109, 110, 111] to assess the quality of windowed reconstruction filters and to derive optimal values for the parameters of Kaiser and Gaussian windows.

An issue of general importance is whether a given filter *interpolates* or only *approximates* the given input function [102, 108]. Approximating filters such as the B-spline can often give pleasing results and suppress noise in the input data [67], but especially in the context of rendering medical volume data function interpolation might be a requirement [102].

In hardware rendering, linear interpolation is usually considered to be a reasonable trade-off between performance and reconstruction quality. High-quality filters are usually only employed when the filtering operation is done in software. However, this thesis demonstrates that general high-quality reconstruction is possible on today's consumer graphics hardware, with cubic reconstruction filters constituting a very attractive alternative to the hardware-native linear interpolation. It has also been shown that tri-cubic filtering achieves significantly better quality than tri-linear interpolation when filtering binary volumes [62], e.g., binary segmentation masks.

For a more thorough general discussion of sampling and reconstruction, and convolution filtering, we refer to the work of Theußl [152].

2.3 Texture Filtering

This section gives an overview of standard texture filtering on consumer graphics hardware since its operation and terminology are important for understanding the modifications we will introduce in later chapters, especially with regard to MIP-mapping.

Filtering and sampling a texture map

In order to apply a texture to a geometric primitive such as a triangle, the texture map must be re-sampled at a basically arbitrary location denoted by texture coordinates. These coordinates are originally specified at the vertices of the primitive and interpolated over its projected area using rational linear interpolation [5, 45].

This re-sampling process consists of two major parts. First, before re-sampling can occur at all, a reconstruction filter must be applied, which maps the discrete function represented by the values in the texture map back to a continuous function. Second, simply re-sampling the texture at a single location is not enough. Theoretically, a *pre-filter* has to be applied in screen space in order to determine the final re-sampled value from many reconstructed samples [47].

Although in theory the re-sampling process is identical for all view points, textures, and pixels, a common approximation in practice is to distinguish between the two cases of *magnification* and *minification* of a texture in order to use different approximations to the ideal re-sampling filter.

Magnification vs. minification

Loosely stated, a texture has to be magnified when the size of a single texel in texture space corresponds to multiple pixels in screen space, whereas it has to be minified when the relation between these sizes is the other way around. In practice, the major difference between magnification and minification is the way in which filtering is performed during re-sampling.

In the case of magnification, the re-sampling filter is dominated by the reconstruction filter, and good results can be achieved without any pre-filter at all. Furthermore, the reconstruction filter is much cheaper to evaluate than the pre-filter, since it always uses a fixed, usually quite low, number of input samples. In graphics hardware, the reconstruction filter is usually either nearest-neighbor or linear interpolation. Our goal in chapter 3 will be to establish higher-order filters as full substitute for these hardware-native filters.

In the case of minification, however, applying the pre-filter is crucial. The major problem is that it requires a potentially unbounded number of input samples and thus in general cannot be evaluated entirely at run time. The most common approach, especially in graphics hardware, is to use some variant of MIP-mapping [178]. Note that although the pre-filter dominates the re-sampling filter in the case of minification, a reconstruction filter still has to be applied in any case as a way for converting the discrete representation in the texture map back into a continuous one. Nevertheless, for minification, a cheaper reconstruction filter such as linear interpolation can be used without significantly compromising quality.

MIP-mapping

MIP-mapping [178] is a very common way to avoid the non-constant, and usually very high, cost of applying the texture re-sampling pre-filter at run time. Basically, the texture map is pre-sampled into a pyramid of textures with successively decreasing resolutions. An approximation to the ideal pre-filter can then be evaluated at run time by simple MIP-map level selection, i.e., choosing a texture image from this pyramid where the size of the pixel under consideration projected into texture space roughly matches the size of a texel [178].

In graphics hardware, this is done for each pixel individually, instead of for an entire triangle. The MIP-map level for each pixel is determined from the partial derivatives of the texture coordinates with respect to screen coordinates. These partial derivatives are usually written in the form of the Jacobian matrix [47, 181]:

$$\begin{pmatrix} \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} \end{pmatrix}$$

These partial derivatives can be determined with the same rational linear interpolation approach that is used for interpolating the texture coordinates themselves [182].

After the MIP-map level has been determined for a given pixel, the same reconstruction filter that is used for the case of magnification can be applied exactly once in order to generate the final output sample. Therefore, MIP-mapping can also be seen as reducing the general case to the case of magnification by selecting an appropriate input texture resolution, and subsequently applying a magnification, i.e., reconstruction, filter.

Graphics hardware also allows to linearly interpolate between two adjacent MIP-map levels. This means applying the reconstruction filter twice, once in each of the two levels, and then linearly interpolating between the two results.

Since the application of the actual reconstruction filter is the same for the cases of magnification and minification, we can extend higher-order magnification filters to also serve as reconstruction filters in the case of minification, as shown in chapter 3.

OpenGL texture filter specification

In OpenGL, the filters used in the cases of magnification and minification of textures are specified separately and are per-texture attributes. For each texture map access, an LOD (level of detail) value is determined from the Jacobian matrix (see above) that is used to distinguish between the cases of magnification and minification [142].

The OpenGL magnification filter specification only specifies the type of reconstruction filter, whereas the minification filter also specifies whether MIP-mapping is enabled or disabled in order to approximate the texture pre-filter.

MIP-mapping is enabled for a given texture by specifying a minification filter of `GL_*_MIPMAP_*`, which specifies both the reconstruction filter used within a MIP-map level, and whether two adjacent levels are interpolated linearly or not (in which case the nearest-neighbor level will be used). The magnification filter (which is either `GL_NEAREST` or `GL_LINEAR`) is only used for filtering the base level (the full-resolution texture image) of the MIP-map pyramid.

Note that even when MIP-mapping is disabled for a texture, the minification filter specification will be used in the case of texture minification, according to the current LOD value. The minification filter in this case is simply also a reconstruction filter, i.e., either `GL_NEAREST` or `GL_LINEAR`, just like the magnification filter.

Texture filtering in the context of this thesis

The filtering algorithms we present in this thesis can be used for texture mapping arbitrary polygonal objects in perspective, filtering static and animated textures, both pre-rendered and procedural, as well as both surface [40], and solid textures [123, 126]. The approach we present can be combined with MIP-mapping [178], which is crucial to using it as full substitute for the usual linear interpolation.

High-quality pre-filtering techniques have been developed for both software [46], and hardware [105] renderers. Hardware pre-filtering usually focuses on extending MIP-mapping for anisotropic filtering in the case of minification, via footprint assembly [105], where several texture lookups at locations approximating the pixel footprint in texture space are combined.

Although most of these methods require explicit hardware support [140], standard MIP-mapping hardware can also be used for anisotropic filtering by accessing several MIP-map levels, and compositing these samples via multi-texturing or multiple rendering passes [118]. Our filtering method also performs filtering by compositing several weighted samples.

Currently, interest in higher quality filtering of textures is resurging, especially in the field of point-based rendering [183].

Hardware convolution

Current graphics hardware has only very limited support for convolution. The OpenGL imaging subset [106] that has been introduced with OpenGL 1.2 can be used for image processing tasks, using 1D and 2D convolutions where the output and input sample grids coincide, and filter kernels are sampled at integer locations only. Building upon the imaging subset, Hopf and Ertl [53] have shown how to perform 3D convolutions for volume processing, and presented research on using graphics hardware for morphological operations [54], and wavelet transforms [55].

Recent graphics hardware features like vertex and pixel shaders can be used for substituting the imaging subset with a faster approach [60, 164], although this is more prone to precision artifacts on architectures without floating point support. The filtering framework we present in this thesis can easily be combined with optional real-time image-processing filters. For these, we combine the standard approach [60] with hierarchical summation in order to reduce precision artifacts.

Recently, non-linear image processing filters have also become possible on GPUs [163], which can be used effectively for edge-preserving filtering, for example. These kinds of filters are especially important as pre-filters before image or volume segmentation [159] is performed.

Reconstruction via hardware convolution

On the most recent graphics hardware architectures, the filter convolution sum can be evaluated entirely in the pixel shader using weights that are determined procedurally by the shader itself on a per-pixel basis [4, 116]. Although this approach is simple to implement, it is highly dependent on the actual filter kernel shape and size, and consumes significant hardware resources in terms of pixel shader instructions and execution time.

Our method for convolution filtering evaluates the filter convolution sum in reverse order than the one that is usually used in software-based convolution, i.e., sample contributions are *distributed* instead of *gathered*, which is also done by all splatting-based volume rendering techniques [177].

2.4 Volume Rendering

The term volume rendering [19, 89, 91] describes a set of techniques for rendering three-dimensional, i.e., volumetric, data. Volume data can be acquired from different sources, e.g., from medical imaging modalities such as computed tomography (CT) or magnetic resonance imaging (MRI) scanners, computational fluid dynamics (CFD), voxelization of objects (e.g., conversion of a triangle mesh into a distance field [145, 148]), or any other data given as a three-dimensional scalar field. Volume data can also be generated synthetically, i.e., procedurally [21], which is especially useful for rendering fluids and gaseous objects, natural phenomena such as clouds, fog, and fire, visualizing molecular structures, or rendering explosions and other effects in 3D computer games.

The two major approaches to volume rendering are *direct volume rendering* (DVR) [19], and *rendering isosurfaces* corresponding to a given scalar iso-value [89, 97].

In direct volume rendering, the scalar field is viewed as being comprised of a cloud of particles of a certain density distribution, which are subsequently assigned optical properties corresponding to an optical model [103], and rendered by solving the volume rendering integral for viewing rays cast into the volume [90, 91, 103]. Especially important in the context of this thesis is direct volume rendering using texture mapping hardware [11, 17, 18, 133]. Another fast alternative to ray casting is the shear-warp factorization of the viewing transform [80]. An overview and comparison of different methods for direct volume rendering has been presented by Meißner et al. [107].

On the other hand, isosurfaces can also be rendered directly from the volume without any intermediate geometric representation [89, 174], or extracted as an explicit polygonal mesh [74, 97]. Figure 2.7 shows a comparison of the same volume data set rendered with direct volume rendering and direct isosurfacing, respectively.

A fundamental concept in volume rendering is the notion of a *transfer function* [68], which assigns optical properties such as color and opacity to scalar data values, and thus determines how different structures embedded in the volume appear in the final image. That is, transfer functions perform the two tasks of identifying different objects via *classification* [91], and subsequently assigning *optical properties* [23] to these objects.

Although volumetric data can be difficult to visualize and interpret, it is both worthwhile and rewarding to visualize them as 3D entities without falling back to 2D subsets. To summarize succinctly, volume rendering is a very powerful way for visualizing volumetric data and aiding the interpretation process, especially in scientific visualization, and can also be used for rendering high-quality special effects.

Volume Data

In contrast to surface data, which are inherently two-dimensional (even though surfaces are often embedded in three-space), volumetric data are comprised of a three-dimensional scalar field:

$$f(\mathbf{x}) \in \mathbb{R} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^3 \tag{2.3}$$

Although in principle defined over a continuous three-dimensional domain ($\mathbb{R}^3$), in the context of volume rendering this scalar field is stored as a 3D array of values, where each of these values is obtained by sampling the continuous domain at a discrete location. The individual scalar data values constituting the sampled volume are referred to as *voxels* (volume

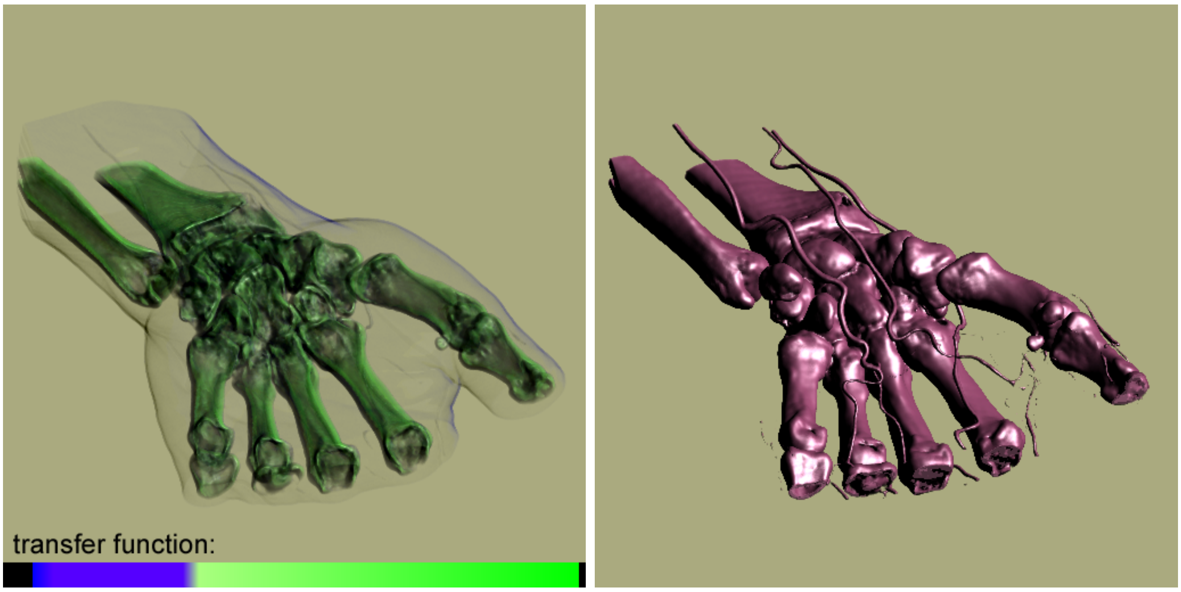


Figure 2.7: Volume data acquired from a CT scan of a human hand. The same view has been rendered with direct volume rendering (left), and direct isosurface rendering (right).

elements), analogously to the term pixels used for denoting the atomic elements of discrete two-dimensional images.

Although imagining voxels as little cubes is convenient and helps to visualize the immediate vicinity of individual voxels, it is more accurate to identify each voxel with a sample obtained at a single infinitesimally small point in $\mathbb{R}^3$. In this model, the volumetric function is only defined at the exact sampling locations. From this collection of discrete samples, a continuous function that is once again defined for all locations in $\mathbb{R}^3$ (or at least the subvolume of interest), can be obtained through reconstruction (section 2.2).

Direct Volume Rendering

Direct volume rendering (DVR) methods [19, 103] create images of an entire volumetric data set, without concentrating on, or even explicitly extracting, surfaces corresponding to certain features of interest, e.g., iso-contours. In order to do so, direct volume rendering requires an *optical model* for describing how the volume emits, reflects, scatters, or occludes light [103]. Different optical models that can be used for direct volume rendering are described in more detail below.

In general, direct volume rendering maps the scalar field constituting the volume to optical properties such as color and opacity, and integrates the corresponding optical effects along viewing rays into the volume, in order to generate a projected image directly from the volume data. The corresponding integral is known as the *volume rendering integral* [19], which is described in more detail below. Naturally, under real-world conditions this integral is solved numerically.

For real-time volume rendering, the *emission-absorption* optical model [103] is usually used, in which a volume is viewed as being comprised of particles at a certain density that are only able to emit and absorb light. In this case, the scalar data constituting the volume is said to denote the density of these particles. Mapping to optical properties is achieved via a transfer function, the application of which is also known as classification. Basically, a transfer function is a lookup table that maps scalar density values to RGBA values, which subsume both the emission (RGB), and the absorption (A) of the optical model. Additionally, the volume can be shaded according to the *illumination* from external light sources.

Optical models

Although most direct volume rendering algorithms, specifically real-time methods, consider the volume to consist of particles at a certain density, and map these densities more or less directly to RGBA information, which is subsequently processed as color and opacity for alpha blending, the underlying physical background is subsumed in an optical model. More sophisticated models than the ones usually used for real-time rendering also include support for scattering of light among particles of the volume itself, and account for shadowing effects.

The most important optical models for direct volume rendering are described in a survey paper by Max [103], and we only briefly summarize these models here:

- **Absorption only.** The volume is assumed to consist of cold, perfectly black particles that absorb all the light that impinges on them. They do not emit, or scatter light.
- **Emission only.** The volume is assumed to consist of particles at a certain density that only emit light, but do not absorb any.
- **Absorption plus emission.** This optical model is the most common one in direct volume rendering. Particles emit light, and occlude, i.e., absorb, incoming light. However, there is no scattering or indirect illumination.
- **Scattering and shading/shadowing.** This model includes scattering of illumination that is external to a voxel. Light that is scattered can either be assumed to impinge unimpeded from a distant light source, or it can be shadowed by particles between the light and the voxel under consideration.
- **Multiple scattering.** This sophisticated model includes support for incident light that has already been scattered by multiple particles.

In this thesis, we are concerned with rendering volumes that are defined on rectilinear grids, using an emission-absorption model together with local illumination for rendering, and do not consider complex lighting situations and effects like single or multiple scattering. However, real-time methods taking such effects into account are currently becoming available [42, 71, 72].

The volume rendering integral

All direct volume rendering algorithms share the property that they evaluate the volume rendering integral, which integrates optical effects such as color and opacity along viewing rays cast into the volume, even if no explicit rays are actually employed by the algorithm. Ray casting could be seen as being the “most direct” numerical method for evaluating this integral. More details are covered below, but here it suffices to view ray casting as a process that, for each pixel in the image to render, casts a single ray from the eye through the pixel’s center into the volume, and integrates the optical properties obtained from the encountered volume densities along the ray.

Note that this general description assumes both the volume and the mapping to optical properties to be continuous. In practice, of course, the evaluation of the volume rendering integral is usually done numerically, together with several additional approximations, and the integration operation becomes a simple summation. Remember that the volume itself is also described by a collection of discrete samples, and thus interpolation, or filtering, for reconstructing a continuous volume has to be used in practice, which is also only an approximation.

We denote a ray cast into the volume by $\mathbf{x}(t)$, and parameterize it by the distance t to the eye. The scalar value corresponding to this position on a ray is denoted by $s(\mathbf{x}(t))$. Since we employ an emission-absorption optical model, the volume rendering integral we are using integrates *absorption coefficients* $\tau(s(\mathbf{x}(t)))$ (accounting for the absorption of light), and *colors* $c(s(\mathbf{x}(t)))$ (accounting for light emitted by particles) along a ray. The volume rendering integral can now be used to obtain the integrated "output" color C, subsuming both color (emission) and opacity (absorption) contributions along a ray up to a certain distance D into the volume:

$$C = \int_0^D c\Big(s\big(\mathbf{x}(t)\big)\Big) e^{-\int_0^t \tau\left(s\left(\mathbf{x}(t')\right)\right) dt'} dt \tag{2.4}$$

Note that for brevity we neglect the contribution of the color of the background, which can easily be incorporated as well [103]. This integral can be understood more easily by looking at different parts individually:

- In order to obtain the color for a pixel (C), we cast a ray into the volume and perform integration along it ($\int_0^D dt$), i.e., for all locations $\mathbf{x}(t)$ along this ray.
- It is sufficient if the integration is performed until the ray exits the volume on the other side, which happens after a certain distance D, where $t = D$.
- The color contribution of the volume at a certain position $\mathbf{x}(t)$ consists of the color emitted there, $c(s(\mathbf{x}(t)))$, multiplied by the cumulative (i.e., integrated) absorption up to the position of emission. The cumulative absorption for that position $\mathbf{x}(t)$ is $\mathbf{exp}(-\int_0^t \tau(s(\mathbf{x}(t'))) \, dt')$.

In practice, this integral is evaluated numerically through either back-to-front or front-to-back compositing (i.e., alpha blending) of samples along the ray, which is most easily illustrated in the method of *ray casting*.

Ray casting

Ray casting [89, 90] is a method for direct volume rendering, which can be seen as straightforward numerical evaluation of the volume rendering integral (equation 2.4). For each pixel in the image, a single ray is cast into the volume (assuming super-sampling is not used). At equispaced intervals along the ray (the sampling distance), the discrete volume data is resampled, usually using tri-linear interpolation as reconstruction filter. That is, for each resampling location, the scalar values of eight neighboring voxels are weighted according to their distance to the actual location for which a data value is needed. After resampling, the scalar data value is mapped to optical properties via a lookup table, which yields an

RGBA value for this location within the volume that subsumes the corresponding emission and absorption coefficients [89], and the volume rendering integral is approximated via alpha blending in back-to-front or front-to-back order.

We will now briefly outline why the volume rendering integral can conveniently be approximated with alpha blending. First, the cumulative absorption up to a certain position $\mathbf{x}(t)$ along the ray, from equation 2.4,

$$e^{-\int_0^t \tau\left(s\left(\mathbf{x}(t')\right)\right)\, dt'} \tag{2.5}$$

can be approximated by (denoting the distance between successive resampling locations with d):

$$e^{-\sum_{i=0}^{\lfloor t/d \rfloor} \tau\left(s\left(\mathbf{x}(id)\right)\right) d} \tag{2.6}$$

The summation in the exponent can immediately be substituted by a multiplication of exponentiation terms:

$$\prod_{i=0}^{\lfloor t/d \rfloor} e^{-\tau\left(s\left(\mathbf{x}(id)\right)\right) d} \tag{2.7}$$

Now, we can introduce the opacity values A “well-known” from alpha blending, by defining

$$A_i \;=\; 1 - e^{-\tau\left(s\left(\mathbf{x}(id)\right)\right) d} \tag{2.8}$$

and rewriting equation 2.7 as:

$$\prod_{i=0}^{\lfloor t/d \rfloor} (1 - A_i) \tag{2.9}$$

This allows us to use A_i as an approximation for the absorption of the i-th ray segment, instead of absorption at a single point.

Similarly, the color (emission) of the i-th ray segment can be approximated by:

$$C_i \;=\; c\left(s\left(\mathbf{x}(id)\right)\right) d \tag{2.10}$$

Having approximated both the emissions and absorptions along a ray, we can now state the approximate evaluation of the volume rendering integral as (denoting the number of samples by $n = \lfloor D/d \rfloor$):

$$C_{approx} \;=\; \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_j) \tag{2.11}$$

Equation 2.11 can be evaluated iteratively by *alpha blending* in either back-to-front, or front-to-back order.

Alpha blending

The following iterative formulation evaluates equation 2.11 in back-to-front order by stepping i from $n-1$ to 0:

$$C'_i = C_i + (1 - A_i) C'_{i+1} \tag{2.12}$$

A new value C'_i is calculated from the color C_i and opacity A_i at the current location i, and the composited color C'_{i+1} from the previous location $i+1$. The starting condition is $C'_n = 0$. This equation amounts to the well-known *over operator* for image compositing [130].

Note that in all blending equations, we are using *opacity-weighted colors* [179], which are also known as *associated colors* [6]. An opacity-weighted color is a color that has been pre-multiplied by its associated opacity. This is a very convenient notation, and especially important for interpolation purposes. It can be shown that interpolating color and opacity separately leads to artifacts, whereas interpolating opacity-weighted colors achieves correct results [179].

The following alternative iterative formulation evaluates equation 2.11 in front-to-back order by stepping i from 1 to n:

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{2.13}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{2.14}$$

New values C'_i and A'_i are calculated from the color C_i and opacity A_i at the current location i, and the composited color C'_{i-1} and opacity A'_{i-1} from the previous location $i-1$. The starting condition is $C'_0 = 0$ and $A'_0 = 0$.

Note that front-to-back compositing requires tracking alpha values, whereas back-to-front compositing does not. In a hardware implementation, this means that *destination alpha* must be supported by the frame buffer (i.e., an alpha valued must be stored in the frame buffer, and it must be possible to use it as multiplication factor in blending operations), when front-to-back compositing is used. However, since the major advantage of front-to-back compositing is an optimization commonly called *early ray termination*, where the progression along a ray is terminated as soon as the cumulative alpha value reaches 1.0, and this cannot easily be done in hardware alpha blending, hardware volume rendering usually uses back-to-front compositing.

GPU-based ray casting

In addition to texture-based volume rendering using slices, which is described in detail in section 2.5, direct ray casting has recently also become possible on GPUs. An early approach [175] used graphics hardware only to obtain a first-hit image of viewing rays with the volume, and proceeded with standard ray casting on the CPU. More recent approaches [78, 137] perform full ray casting on the GPU by tracking three-dimensional ray positions in volume space on a per-pixel basis.

Transfer Functions

Transfer functions for classification and mapping of volume densities to optical properties are an extremely important part of direct volume rendering. The term classification, which identifies objects in a volume, is often used to denote a pure opacity transfer function, with additional optical properties specified separately. However, the most common type of transfer function is simply a one-dimensional table in the domain of volume densities that stores RGBA values for colors and opacities [91].

Separable multi-dimensional transfer functions have also been in use for a long time, e.g., incorporating gradient magnitude as second dimension in addition to volume density [89].

Recently, more general multi-dimensional transfer functions [66, 68] have become a very important tool for distinguishing different objects contained in a volume. They can be used in interactive volume rendering on graphics hardware [70], which allows to use intuitive user interfaces with real-time feedback for specification of the transfer function [69] and high-quality results [23]. Multi-dimensional transfer functions become also an issue when the volume does not simply contain densities but color values obtained by taking photographs [20]. Transfer functions in the domain of principal curvature magnitudes have also proven to be very powerful for highlighting and identifying different shape structures [50, 52, 139] and non-photorealistic volume rendering [67].

Recently, very high-dimensional transfer functions have been used to incorporate the spatial domain into the transfer function [158], and to render multi-variate volume data interactively on consumer graphics hardware with transfer functions composed of Gaussians [73].

A very important problem of transfer functions is their specification in a way that allows to see the desired objects and visualize the desired information. A very powerful approach to semi-automatically generating transfer function is to use statistical and differential measures such as histogram volumes and first and second order partial derivatives [66, 68]. Image-guided transfer function specification can help the user select desired characteristics, e.g., employing genetic algorithms [48], or presenting a series of thumbnail images [75, 100], or spreadsheet-like interfaces [61]. Another idea is to adapt existing transfer functions to a new data set [134]. It is also possible to generate meaningful volume renderings without any transfer function at all [15]. An interesting evaluation of different approaches to transfer function specification has been presented at the Transfer Function Bake-Off [127, 128].

A problem similar to the specification of a full transfer function is the detection of meaningful iso-values. The Contour Spectrum [2] provides a visualization of meaningful iso-values in unstructured volume data and thus helps with the specification of isosurfaces by the user. Surface integrals over an isosurface can be used in order to determine the correspondence of a given iso-value to the significance of the corresponding boundary [125]. Statistical measures are also a powerful approach to detecting meaningful iso-values [150].

Pre-integrated volume rendering [24] substitutes a one-dimensional transfer function texture by a two-dimensional pre-integration table, and decouples the frequencies contained in the scalar volume from the frequencies in the transfer function. It thus allows to achieve high-quality results even with low sampling rates.

Maximum Intensity Projection

Maximum intensity projection (MIP) is a variant of direct volume rendering, where, instead of compositing optical properties, the maximum value encountered along a ray is used to determine the color of the corresponding pixel. An important application area of such a rendering mode are medical data sets obtained by MRI (magnetic resonance imaging) scanners. Such data sets usually exhibit a significant amount of noise that can make it hard to extract meaningful isosurfaces, or define transfer functions that aid the interpretation. When MIP is used, however, the fact that within angiography data sets the data values of vascular structures are higher than the values of the surrounding tissue, can be exploited easily for visualizing them. In graphics hardware, MIP can be implemented by using a maximum operator when blending into the frame buffer, instead of standard alpha blending. The corresponding OpenGL extension is described in Appendix B.

Isosurface Rendering

In the context of volume rendering, the term *isosurface* denotes a contour surface extracted from a volume that corresponds to a given constant scalar iso-value. Boundary surfaces of regions of the volume that are homogeneous with respect to certain attributes are usually also called isosurfaces. For example, an explicit isosurface could be used to depict a region where the density is above a given threshold.

There is a large amount of research on rendering isosurfaces of volumetric data such as CT or MRI scans, as well as the area of implicit surfaces in general, especially when an implicit is represented by a grid of function samples, e.g., in levelset methods [113].

As the name suggests, an isosurface is usually constituted by an explicit surface. In contrast to direct volume rendering, where no surfaces exist at all, these explicit surfaces are often extracted from the volume data in a pre-process. This is commonly done by using the marching cubes algorithm [97] or one of its variants [74]. These algorithms generate an explicit geometric representation (usually thousands to millions of triangles) for the feature of interest, i.e., the isosurface corresponding to a given iso-value from the volume data.

That is, although isosurfaces are often converted to triangle meshes for rendering [25, 74, 97], this produces very complex models and interactive changes of the iso-value or the volume itself are difficult to deal with. The extraction of isosurfaces can be accelerated by keeping track of blocks that potentially intersect a given isosurface using an interval tree [12, 13], reducing the size of the subvolume that must be considered. Other well-known approaches for speeding up isosurface extraction build on the extrema graph [59], and the span space [93], respectively. Isosurface geometry can also be extracted in a view-dependent manner [92].

However, isosurfaces can also be rendered without the presence of explicit geometry. In this case, we will refer to them as *non-polygonal isosurfaces* or, more generally, *direct isosurface rendering*. The two major approaches for directly rendering isosurfaces are ray casting [3, 89], and sampling ray-surface intersections on graphics hardware [174]. Although implicit surfaces are well-suited for finding guaranteed ray-surface intersections [63], precise computations and high-quality reconstruction are expensive. Hence interactive rates with high-quality or analytic ray-surface intersections and gradients have only been achieved by implementations using clusters [121].

On graphics hardware, non-polygonal isosurfaces can be rendered by exploiting the OpenGL alpha test [174]. In the original approach, the volume is stored as an RGBA volume. Local gradient information is pre-computed and stored in the RGB channels, and the volume density itself is stored in the alpha channel. The density in conjunction with alpha testing is used in order to select pixels where the corresponding ray pierces the isosurface, and the gradient information is used as “surface normal” for shading. The concept of pre-integration [24] can also be employed for rendering high-quality isosurfaces with a moderate sampling rate, since it is able to deal well with high frequencies in the transfer function. On current GPUs, it is possible to compute high-quality ray-isosurface intersections with tri-cubic reconstruction [144], which can be combined very effectively with *deferred shading* and on-the-fly computation of high-quality gradients and other differential properties, as illustrated in chapter 5 together with their application for non-photorealistic techniques.

Naturally, it is not only important how to render an isosurface, but also which iso-value corresponds to the structure of interest. This is not a trivial problem, but over the years powerful methods to tackle it have been developed. See the section on transfer functions above for related work on the specification of meaningful iso-values.

Level-sets and signed distance fields

If the volume data have not been scanned directly, signed distance fields are a natural choice as input for isosurface rendering. Signed distance fields can be generated quickly on current graphics hardware [145, 148].

Levelset methods change the distance fields dynamically and have many powerful applications such as surface editing and processing operators [113], and surface deformations [157]. Narrow-band levelset computations have recently become possible on GPUs [84, 85, 86, 87].

Computing and visualizing surface curvature

In addition to reconstructing an isosurface, computing implicit surface curvature for it has many interesting applications [67]. For example, the space of principal curvature magnitudes is intuitive for shape depiction [50, 52]. Incidentally, the Hessian matrix that is fundamental to computing implicit surface curvature can also be used effectively in order to determine salient parts of a volume [51], e.g., for volume compression or progressive transmission. Curvature information can also be used as a basis for non-photorealistic techniques, e.g., through ridge and valley lines [57, 67].

Principal curvature directions can be visualized effectively by 3D line integral convolution [58]. A very interesting alternative is to advect dense noise textures on the surface [162], which can be done entirely in image space [82, 83].

An interesting application of visualizing surface curvature is to visualize the process of surface smoothing [67], e.g., anisotropic surface smoothing [149].

Non-Photorealistic Volume Rendering

Non-photorealistic rendering is a powerful technique for enhancing the perception of different structures of interest. These techniques usually operate on surfaces [28, 29, 146], but can be adapted to volume rendering [135].

Recently, purely volumetric non-photorealistic models have also been developed. Contours can be rendered without an explicit notion of surfaces [16], and stippling techniques can be applied on a per-voxel basis [99]. Pen-and-ink rendering styles can be used to generate effective non-photorealistic volume renderings [155]. Hatching strokes and silhouette lines for volume illustration can be rendered interactively on graphics hardware [114]. Curvature information (see above) is also a very powerful basis for a variety of non-photorealistic techniques, e.g., through ridge and valley lines [57, 67] and for rendering curvature-controlled contours [67].

If no segmentation information is present, multiple rendering passes with one transfer function each and non-photorealistic shading can be used in order to enhance perception of individual objects [98].

Clipping and Segmented Data

The topic of image and volume segmentation is a huge area on its own [159]. In this thesis, segmentation information as simply treated as additional a priori input data that are already available for a given data set.

When rendering segmented volumes, the filtering quality of object boundaries is crucial [154]. Even linear filtering of segmentation data is not directly possible on graphics hardware when more than two objects have been segmented, since object IDs cannot be interpolated directly. Chapter 4 shows how this can be done very efficiently on current graphics hardware. Ultimately, rendering segmented data sets can be viewed as being composed of multiple individual volumetric clipping problems. Recent work has shown how to achieve high-quality clipping in graphics hardware [168, 169], which can also be combined with pre-integrated classification by adjusting the lookup into the pre-integration table accordingly [137]. However, it is not trivial to apply clipping approaches to the rendering of segmented data as soon as the volume contains more than two objects and high-quality results and a minimal number of rendering passes are desired.

A powerful approach to rendering segmented data with improved perception of individual objects is the concept of two-level volume rendering [43, 44], which combines different compositing modes on a local per-object basis with a single global compositing mode. Chapter 4 also illustrates high-quality two-level volume rendering on graphics hardware.

Large Data

A major drawback of current consumer graphics hardware is the limited amount of on-board texture memory (currently only up to 256MB, which is also shared with frame buffer memory). Various approaches have been developed to deal with large data. The volume is usually subdivided into blocks, e.g., an octree structure [81], where the issue of avoiding artifacts at block boundaries is crucial [167].

Data compression is a natural approach to dealing with large data sets, e.g., using wavelet techniques [32, 33]. Decompression of volume data can also be done entirely on the GPU [141], which is especially useful for rendering time-dependent data [141].

A fundamental concept used by many techniques for dealing with large volume data is rearranging smaller sub-textures in larger textures [77]. In addition to dealing with large data, such blocking strategies can also be used effectively for optimization strategies such as avoiding rasterization of empty texture areas [94], empty space skipping [95, 96], and early ray termination [96]. Block hierarchies have also been employed for adaptive re-sampling of unstructured grids into rectilinear textures [88].

The size of volume data sets is naturally also an issue for isosurface geometry extraction algorithms [10, 12].

Structured and Unstructured Grids

In addition to volume data given on Cartesian or rectilinear grids, volume data are often also specified on structured or unstructured grids, especially in computational fluid dynamics (CFD). These grid types consist of a collection of cells, such as tetrahedral or hexahedral cells, that are connected in an either completely random topology (*unstructured grids*), or in a structure that is topologically still equivalent to a rectilinear grid, but whose vertices do not lie on lattice points (*structured grids*). Although the cells themselves are usually convex, the cell complex (the volumetric mesh) need not be, which leads to a variety of problems in handling them in a consistent manner [76].

Common methods for rendering such meshes project cell faces, which can be done efficiently on graphics hardware [138], although it is still much more computationally expensive than rendering Cartesian data. Recently, it has also become possible to perform ray casting of unstructured grids on GPUs [166].

Another approach to dealing with non-rectilinear grids is to re-sample them onto a rectilinear lattice before rendering [165, 176], whereafter they can be handled by all methods for this volume type. In order to be able to handle large data sizes, re-sampling can also be done hierarchically and on-demand [88].

Point-Based Volume Rendering

The area of point-based rendering [183] has recently been a very active area of research. In addition to approaches for volume rendering [14, 56, 112] based on traditional splatting [177], new point-based representations have been developed recently [31, 132, 173].

Flow Advection

Another interesting area of volume visualization is the visualization of flow by advecting dense noise patterns [161], often also combined with the injection of dye [171]. Traditional approaches work in two-dimensional image space [161, 170], but have also been extended to three-dimensional vector fields [172]. Recent developments visualize flow on surfaces by performing advection in image space [82, 83, 162].

In the context of this thesis, flow advection algorithms are important for visualizing the principal curvature directions of isosurfaces that are extracted on-the-fly in real-time, as shown in chapter 5.

2.5 Texture-Based Volume Rendering

Direct volume rendering [19] evaluates the volume rendering integral for viewing rays cast into the volume. The fundamental operation for a numerical evaluation of this integral is re-

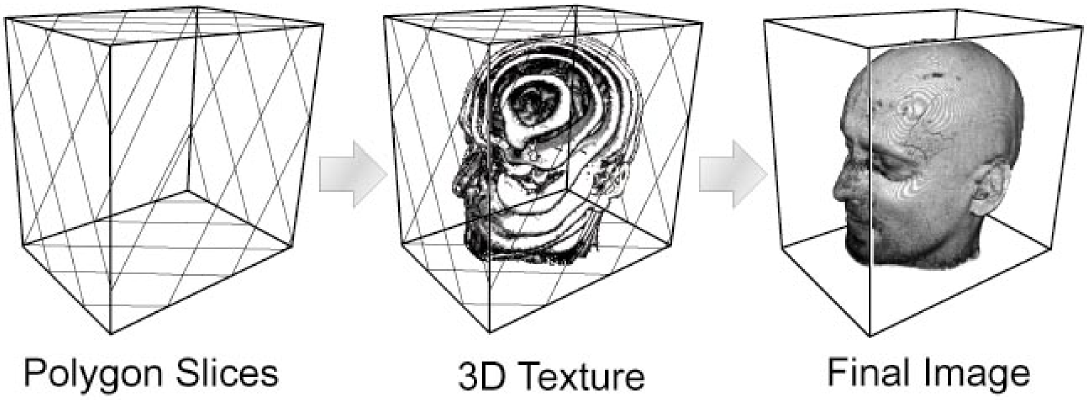


Figure 2.8: View-aligned slices used as proxy geometry with 3D texture mapping [23].

sampling the volume, which can be done at interactive rates with the use of texture mapping hardware, the fundamentals of which will be illustrated in this section.

The easiest way to re-sample a volume in texture mapping hardware is to store it in a 3D texture and slice the volume with polygons and tri-linear interpolation [11, 17, 160]. An alternative approach is to use 2D textures with bi-linear interpolation and an additional interpolation step between adjacent slices [133]. This approach allows to attain tri-linear interpolation with simple 2D texture mapping. The number of slices necessary for high-quality results can be reduced drastically by considering two adjacent slices as constituting a single slab and compensating for non-linear transfer function changes within each slab via a lookup into a pre-integrated transfer function table [24], which effectively reduces aliasing artifacts due to under-sampling during slicing.

This section gives an overview of how a volume is re-sampled and rendered in traditional texture-based volume rendering based on slicing the volume [11, 17]. The approaches outlined in this section re-sample and render a volume in *object-order*. However, *image-order* volume rendering via direct ray casting on programmable graphics hardware has recently also become a viable alternative [78, 137].

Re-sampling a volume

As illustrated earlier in this chapter, the most fundamental operation in volume rendering is sampling the volumetric data. Since these data are already discrete, the sampling task performed during rendering is actually a *re-sampling* task, i.e., re-sampling sampled volume data from one set of discrete locations to another. In order to render a high-quality image of the entire volume, these re-sampling locations have to be chosen carefully, followed by mapping the obtained values to optical properties, such as color and opacity, and compositing them in either back-to-front or front-to-back order.

Ray casting is probably the simplest approach for accomplishing this task. Because it casts rays from the eye through image plane pixels back into the volume, ray casting is usually called an image-order approach. That is, each ray is cast into the volume, which is then re-sampled at – usually equispaced – intervals along that ray. The values obtained via

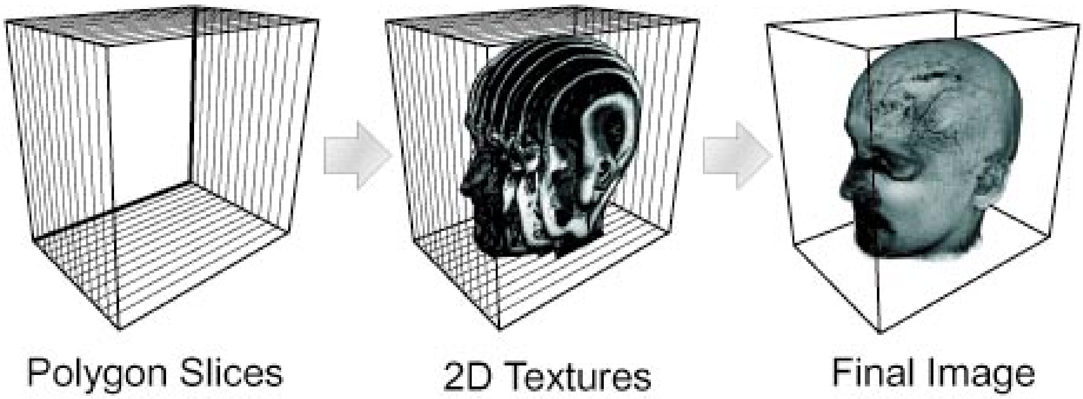


Figure 2.9: Object-aligned slices used as proxy geometry with 2D texture mapping [23].

re-sampling are mapped to color and opacity, and composited in order along the ray (from the eye into the volume, or from behind the volume toward the eye) via alpha blending.

Texture mapping operations basically perform a similar task, i.e., re-sampling a discrete grid of texels to obtain texture values at locations that do not coincide with the original grid. Thus, texture mapping in many ways is an ideal candidate for performing repetitive re-sampling tasks. Compositing individual samples can easily be done by exploiting hardware alpha blending. The major question with regard to hardware-accelerated volume rendering is how to achieve the same – or a sufficiently similar – result as compositing samples taken along a ray cast into the volume.

The major way in which hardware texture mapping can be applied to volume rendering is to use an object-order approach, instead of the image-order approach of ray casting. The re-sampling locations are generated by rendering *proxy geometry* with interpolated texture coordinates (usually comprised of slices rendered as texture-mapped quads), and compositing all the parts (slices) of this proxy geometry from back to front via alpha blending. The volume data are stored in one to several textures of two or three dimensions, respectively. For example, if only a density volume is required, it can be stored in a single 3D texture, where a single texel corresponds to a single voxel. Alternatively, volume data can be stored in a stack of 2D textures, each of which corresponds to an axis-aligned slice through the volume.

By rendering geometry mapped with these textures, the original volume can be re-sampled at specific locations, blending the generated fragments with the previous contents of the frame buffer. Such an approach is called object-order, because the algorithm does not iterate over individual pixels of the image plane, but over parts of the "object," i.e., the volume itself. That is, these parts are usually constituted by slices through the volume, and the final result for each pixel is only available after all slices contributing to this pixel have been processed.

Proxy geometry

In all approaches rendering volumetric data directly, i.e., without any geometry that has been extracted along certain features (e.g., polygons corresponding to an isosurface, generated by a variant of the marching cubes algorithm [97]), there exists no geometry at all, at least not per se. However, geometry is the only thing graphics hardware with standard texture mapping capabilities is actually able to render. In this sense, all the fragments and ultimately pixels rendered by graphics hardware are generated by rasterizing geometric primitives, in most cases triangles. That is, sampling a texture has to take place through such primitives specified by their vertices.

The collective geometry used for obtaining all re-sampling locations needed for sampling the entire volume is commonly called *proxy geometry*, since it has no inherent relation to the data contained in the volume itself, and exists solely for the purpose of generating re-sampling locations, and subsequently sampling texture maps at these locations [11, 17].

The conceptually simplest example of proxy geometry is a set of *view-aligned slices* (quads that are parallel to the viewport, usually also clipped against the bounding box of the volume, see figure 2.8), with 3D texture coordinates that are interpolated on these slices, and ultimately used to sample a single 3D texture map at the corresponding locations. However, 3D texture mapping is not supported by all consumer graphics hardware, and even on hardware that does support it, 3D textures incur a performance penalty in comparison to 2D textures. This penalty is mostly due to the tri-linear interpolation used when sampling a 3D texture map, as opposed to bi-linear interpolation for sampling a 2D texture map.

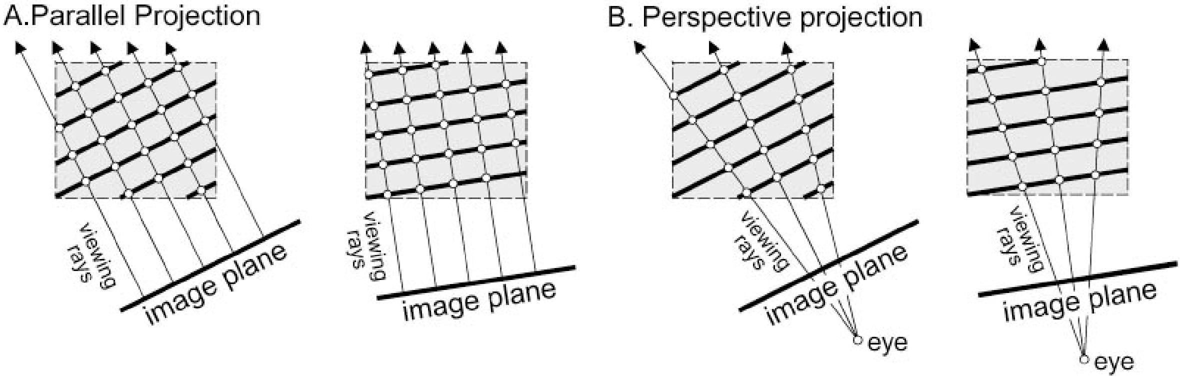


Figure 2.10: Sampling locations on view-aligned slices for parallel (A), and perspective projection (B), respectively [23].

One of the most important things to note about proxy geometry is that it is intimately related to the kind of texture mapping (2D or 3D) used. When the orientation of slices with respect to the original volume data (i.e., the texture) can be arbitrary, 3D texture mapping is mandatory, since a single slice would have to fetch data from several different 2D textures. If, however, the proxy geometry is aligned with the original volume data, texture fetch operations for a single slice can be guaranteed to stay within the same 2D texture. In this case, the proxy geometry is comprised of a set of *object-aligned slices* (see figure 2.9), for which 2D texture mapping capabilities suffice. Below, different kinds of proxy geometry and the corresponding re-sampling approaches are described in more detail.

3D-Textured View-Aligned Slices

In many respects, 3D-textured view-aligned slices are the simplest kind of proxy geometry (see figure 2.8). In this case, the volume is stored in a single 3D texture, and 3D texture coordinates are interpolated on proxy geometry polygons. These texture coordinates are then used directly for indexing the 3D texture map at the corresponding location, and thus re-sampling the volume.

The big advantage of 3D texture mapping is that it allows slices to be oriented arbitrarily with respect to the 3D texture domain, i.e., the volume itself. Thus, it is natural to use slices aligned with the viewport, since such slices closely mimic the ray casting algorithm. They offer constant distance between samples for orthogonal projection and all viewing directions, see figure 2.10(A). Since the graphics hardware is already performing completely general tri-linear interpolation within the volume for each re-sampling location, proxy slices are not bound to original slices at all. Thus, the number of slices can easily be adjusted on-the-fly and without any restrictions, or the need for separately configuring inter-slice interpolation.

In the case of perspective projection, the distance between successive samples is different for adjacent pixels, however, which is depicted in figure 2.10(B). If the artifacts caused by a not entirely accurate compensation for sampling distance is deemed noticeable, spherical shells can be employed instead of planar slices.

Discussion

The biggest advantage of using view-aligned slices and 3D textures for volume rendering is that tri-linear interpolation can be employed for re-sampling the volume at arbitrary locations. Apart from better image quality than with using bi-linear interpolation, this allows to render slices with arbitrary orientation with respect to the volume, which makes it possible to maintain a constant sampling rate for all pixels and viewing directions. Additionally, a single 3D texture suffices for storing the entire volume.

The major disadvantage of this approach is that it requires hardware-native support for 3D textures, which is not yet widely available, and tri-linear interpolation is also significantly slower than bi-linear interpolation, due to the requirement for using eight texels for every single output sample, and texture fetch patterns that decrease the efficiency of texture caches.

2D-Textured Object-Aligned Slices

If only 2D texture mapping capabilities are used, the volume data must be stored in several two-dimensional texture maps. A major implication of the use of 2D textures is that the hardware is only able to re-sample two-dimensional subsets of the original volumetric data. The proxy geometry in this case is a stack of planar slices, all of which are required to be aligned with one of the major axes of the volume (either the x, y, or z axis), mapped with 2D textures, which in turn are re-sampled by the hardware-native bi-linear interpolation [11]. The reason for the requirement that slices be aligned with a major axis is that each time a slice is rendered, only two dimensions are available for texture coordinates, and the third coordinate must therefore be constant. Also, bi-linear interpolation would not be sufficient for re-sampling otherwise. Now, instead of being used as an actual texture coordinate, the third coordinate selects the texture to use from the stack of slices, and the other two coordinates become the actual 2D texture coordinates used for rendering the slice. Rendering proceeds from back to front, blending one slice on top of the other (see figure 2.9).

Although a single stack of 2D slices can store the entire volume, one slice stack does not suffice for rendering. When the viewpoint is rotated about the object, it would be possible to see between individual slices, which cannot be prevented with only one slice stack. The solution for this problem is to actually store three slice stacks, one for each of the major

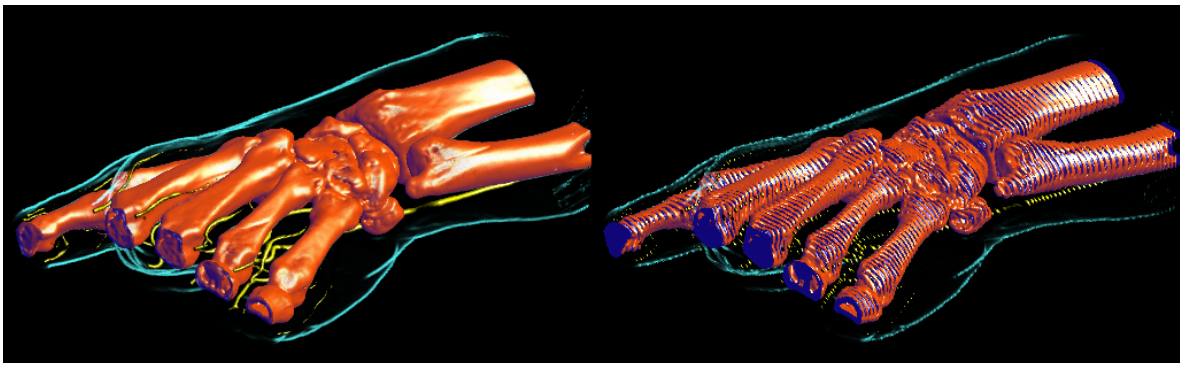


Figure 2.11: Rendering a volume by compositing a stack of 2D texture-mapped slices in back-to-front order. If the number of slices is too low, they become visible as artifacts (right).

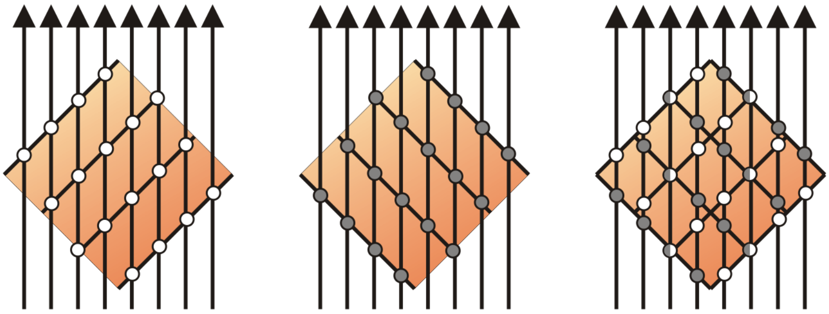


Figure 2.12: The location of sampling points changes abruptly (right), when switching from one slice stack (left), to the next (center) [23].

axes. During rendering, the stack with slices most parallel to the viewing direction is chosen. Under-sampling typically occurs most visibly along the major axis of the slice stack currently in use, which can be seen in figure 2.11. Additional artifacts become visible when the slice stack in use is switched from one stack to the next. The reason for this is that the actual locations of sampling points change abruptly when the stacks are switched, which is illustrated in figure 2.12. To summarize, an obvious drawback of using object-aligned 2D slices is the requirement for three slice stacks, which consume three times the texture memory a single 3D texture would consume. When choosing a stack for rendering, an additional consideration must also be taken into account: After selecting the slice stack, it must be rendered in one of two directions, in order to guarantee actual back-to-front rendering. That is, if a stack is viewed from the back (with respect to the stack itself), it has to be rendered in reversed order, to achieve the desired result.

Discussion

The biggest advantage of using object-aligned slices and 2D textures for volume rendering is that 2D textures and the corresponding bi-linear interpolation are a standard feature of all 3D graphics hardware architectures, and therefore this approach can practically be implemented anywhere. Also, the rendering performance is extremely high, since bi-linear interpolation requires only a lookup and weighting of four texels for each re-sampling operation.

The major disadvantages of this approach are the high memory requirements, due to the three slice stacks that are required, and the restriction to using two-dimensional, i.e., usually bi-linear, interpolation for texture reconstruction. The use of object-aligned slice stacks also leads to sampling and stack switching artifacts, as well as inconsistent sampling rates for different viewing directions.

2D Slice Interpolation

Figure 2.11 shows a fundamental problem of using 2D texture-mapped slices as proxy geometry for volume rendering. In contrast to view-aligned 3D texture-mapped slices, the number of slices cannot be changed easily, because each slice corresponds to exactly one slice from the slice stack. Furthermore, no interpolation between slices is performed at all, since only bi-linear interpolation is used within each slice. Because of these two properties of that algorithm, artifacts can become visible when there are too few slices, and thus the sampling frequency is too low with respect to frequencies contained in the volume and the transfer function.

In order to increase the sampling frequency without enlarging the volume itself (e.g., by generating additional interpolated slices before downloading them to the graphics hardware), inter-slice interpolation has to be performed on-the-fly by the graphics hardware itself. On the consumer hardware most interesting in the context of this thesis (i.e., NVIDIA GeForce or later, and ATI Radeon 8500 or later), this can be achieved by using two simultaneous textures when rendering a single slice, instead of just one texture, and performing linear interpolation between these two textures [133]. In order to do this, fractional slice positions have to be specified, where the integers correspond to slices that actually exist in the source slice stack, and the fractional part determines the position between two adjacent slices. The number of rendered slices is now independent from the number of slices contained in the volume, and can be adjusted arbitrarily.

For each slice to be rendered, two textures are activated, which correspond to the two neighboring original slices from the source slice stack. The fractional position between these slices is used as weight for the inter-slice interpolation. This method actually performs tri-linear interpolation within the volume. Standard bi-linear interpolation is employed for each of the two neighboring slices, and the interpolation between the two obtained results altogether achieves tri-linear interpolation.

Discussion

The biggest advantage of using object-aligned slices together with on-the-fly interpolation between two 2D textures for volume rendering is that this method combines the advantages of using only 2D textures with the capability of arbitrarily controlling the sampling rate, i.e., the number of slices. Although not entirely comparable to tri-linear interpolation in a 3D texture, the combination of bi-linear interpolation and a second linear interpolation step ultimately allows tri-linear interpolation in the volume. The necessary features of consumer graphics hardware, i.e., multi-texturing with at least two simultaneous textures, and the ability to interpolate between them, have been widely available for several years.

Disadvantages inherent to the use of object-aligned slice stacks still apply, though. For example, the undesired visible effects when switching slice stacks, and the memory consumption of the three slice stacks.

Opacity Correction

In texture-based volume rendering, hardware alpha blending is used to achieve the same effect as compositing samples along rays in ray casting. This alpha blending operation actually is a method for performing a numerical integration of the volume rendering integral. The distance between successive re-sampling locations along a “ray,” i.e., the distance at which the integral is approximated by a summation, most of all depends on the distance between adjacent slices.

The sampling distance is easiest to account for if it is constant for all “rays” (i.e., pixels). In this case, it can be incorporated into the numerical integration in a pre-process, which is usually done by simply adjusting the transfer function lookup table accordingly. In order to adjust opacities (alpha values) stored in the transfer function to correspond to a new sampling

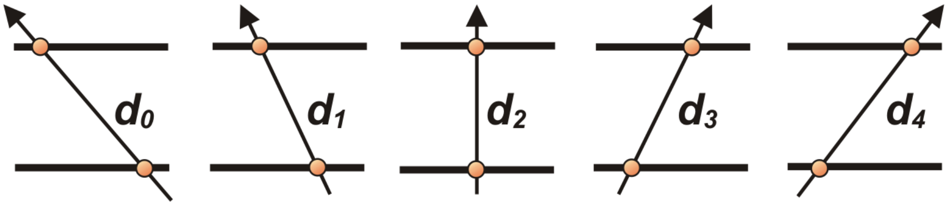


Figure 2.13: In the case of 2D texture-mapped object-aligned slices, the distance between successive sampling points depends on the angle between the viewing rays and the orientation of the slices through the volume [23]. Note that in the case of orthogonal projection, this angle is the same for all viewing rays.

rate given as distance d between two successive samples, i.e., slices, the following approximate formula is often used [91]:

$$\alpha_{\text{new}} = 1 - \left(1 - \alpha_{\text{old}}{}^{\frac{\mathrm{d}_{\text{new}}}{\mathrm{d}_{\text{old}}}}\right) \tag{2.15}$$

In the case of 3D-textured slices (and orthogonal projection), the slice distance is equal to the sampling distance, which is also equal for all "rays" (i.e., pixels). Thus, it can be accounted for entirely in a pre-process. When 2D-textured slices are used, however, the distance between successive samples for each pixel not only depends on the slice distance, but also on the viewing direction. This is shown in figure 2.13 for two adjacent slices. The sampling distance is only equal to the slice distance when the stack is viewed perpendicularly to its major axis. When the view is rotated, the sampling distance increases. For this reason, the lookup table for numerical integration (the transfer function table) has to be updated on each change of the viewing direction.

Slices vs. Slabs

An inherent problem of using slices as proxy geometry is that the number of slices directly determines the (re)sampling frequency, and thus the quality of the rendered result. Especially when high frequencies are contained in the transfer functions employed, the required number of slices can become very high. Thus, even though the number of slices can be increased on-the-fly via interpolation done by the graphics hardware itself, the fill rate demands increase significantly.

A very elegant solution to this problem is to use *slabs* instead of slices, together with pre-integrated classification [24]. A slab is no new geometrical primitive, but simply the space between two adjacent slices. During rendering, this space is properly accounted for, instead of simply rendering infinitesimally thin slices, by looking up the pre-integrated result of the volume rendering integral from the back slice to the front slice in a lookup table, i.e., a texture. Geometrically, a slab can be rendered as a slice with its immediately neighboring slice (either in the back, or in front) projected onto it [24].

Chapter 3

High-Quality Filtering

A very important basic step of volume rendering is the reconstruction of the original continuous signal from sampled data as discussed in section 2.2. In the general case, this reconstruction step is accomplished by filtering the original data via linear convolution with a reconstruction filter.

In texture mapping, or texture-based volume rendering, the input data are given as samples stored in a texture map, and reconstruction (filtering) is usually performed on-the-fly during rendering. In principle, convolution filtering can be performed with arbitrary reconstruction filters. However, the texture filtering modes of current graphics hardware are very limited in order to achieve high performance. The only two filtering modes supported natively for magnification of texture maps by most graphics hardware are *nearest-neighbor interpolation* and *linear interpolation* (including *bi-linear* and *tri-linear* interpolation). The former method is equivalent to convolution with a box filter (figure 2.6, left), and the latter one to convolution with a tent filter (figure 2.6, center).

Unfortunately, linear interpolation achieves only moderate quality in comparison to higher-order filters such as cubic splines, for example. However, although higher-order filters are able to achieve significantly better quality than linear interpolation, they are usually only used for filtering in software rendering due to their computational complexity and the fact that higher-order filters are not supported natively by most graphics hardware.

Contribution

The main contribution of this chapter is a general framework for evaluation of arbitrary convolution filters for real-time high-quality filtering of textures during rendering on standard graphics hardware. Although the focus is on *texture magnification* filters, proper combination with MIP-mapping allows to employ higher-order filters also in the case of *texture minification.* Together, this allows to substitute all major reconstruction tasks in hardware texture mapping pipelines that are usually restricted to nearest-neighbor or linear interpolation by higher-order filters. See figure 3.1 for an overview comparison of linear and higher-order filtering quality for texture magnification and minification, respectively.

The only basic requirement for hardware-accelerated convolution filtering in our framework is support for texture mapping with at least two simultaneous textures (*multi-texturing*) without the need for advanced programmability. However, a higher number of simultaneous textures, as well as the flexible programmability of recent GPU architectures, allow for several optimizations, yielding a family of methods offering different performance and quality trade-offs with respect to required hardware features and filter kernel properties.

Although for many important filter kernels high-quality results can be achieved on graphics hardware only supporting 8-bit precision per color channel in a $[0,1]$ range, arbitrary convolution filters require higher precision and especially range for intermediate results. We treat this issue in section 3.5. Recent GPU architectures have also introduced higher precision of per-pixel computations, including support for full 32-bit floating point operations,

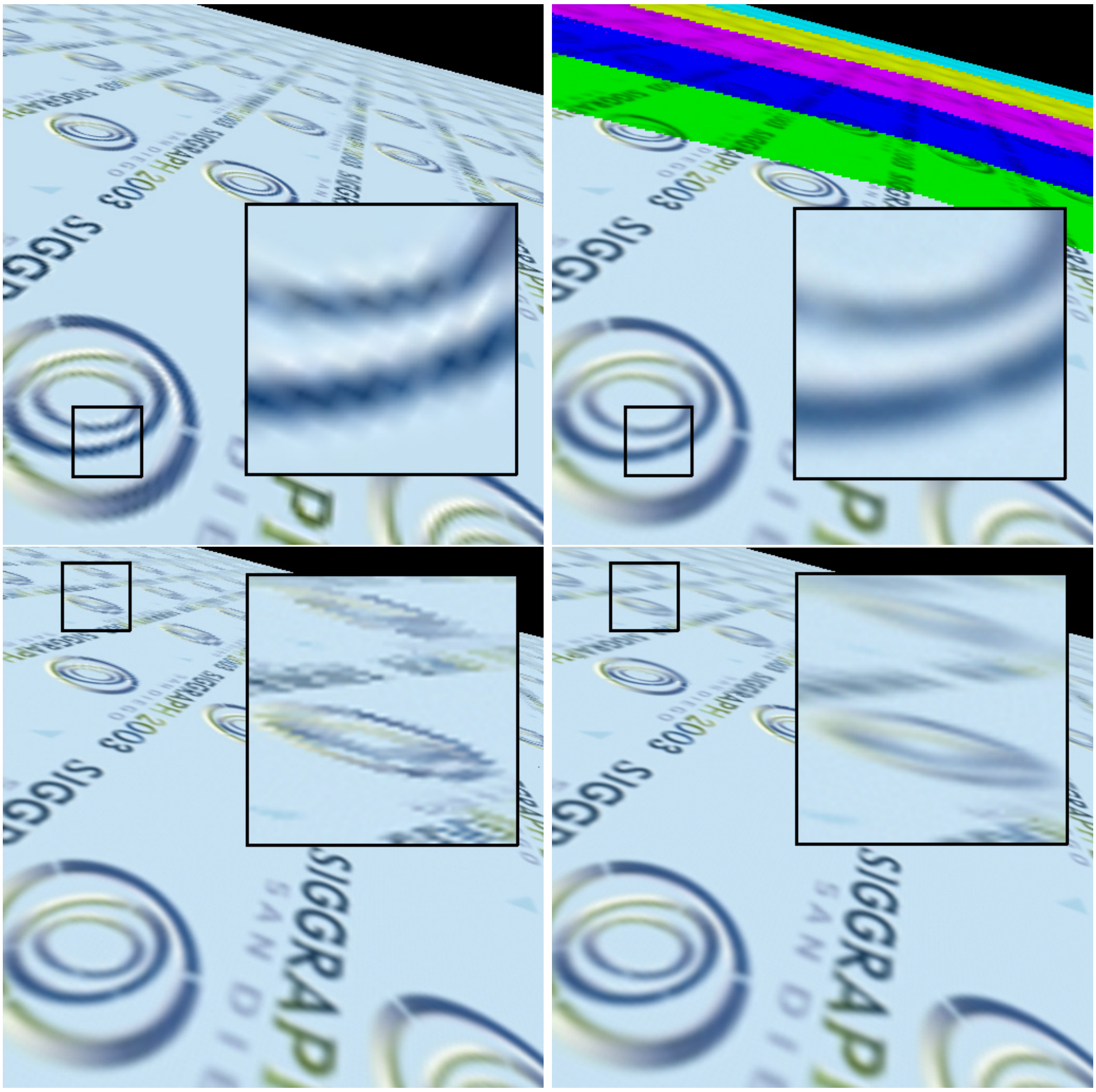


Figure 3.1: Higher-order vs. linear filtering of textures in both the cases of texture magnification and minification, respectively. (top, left) texture magnification using hardware-native linear interpolation; (top, right) texture magnification with a cubic B-spline filter using our framework; MIP-map levels are color-coded in this image; (bottom, left) in lower-resolution MIP-map levels, the cubic filter degenerates to nearest-neighbor interpolation if it is not adapted to the level of texture minification; (bottom, right) cubic filtering in all MIP-map levels, including magnification (base level of MIP-map) and minification (other levels).

which allows our framework to obtain the quality achieved by pure CPU-based floating point filtering, while retaining real-time performance.

In this chapter, we substitute both of the hardware-native texture reconstruction filters used by OpenGL, the magnification and the minification filter, by higher-order filters. This allows us, for example, to implement a `GL_CUBIC` filter for the OpenGL magnification filter specification, and `GL_CUBIC_MIPMAP_NEAREST` and `GL_CUBIC_MIPMAP_LINEAR` filters for the minification filter specification. With regard to graphics hardware, specifically using the OpenGL API, texturing and filtering functionality and terminology we refer to section 2.3.

Organization of this chapter

This chapter is organized as follows. Section 3.1 describes the main algorithms of our filtering framework, focusing on magnification filters. Extensions for filtering MIP-mapped textures with proper application of higher-order filters in the case of minification are described in section 3.2. Applications to texture filtering of surface (2D) and solid (3D) textures are shown in section 3.3, and section 3.4 illustrates how our framework can be used for improving the reconstruction quality in volume rendering. Section 3.5 analyzes the error incurred by filtering on graphics hardware architectures using fixed point color representations instead of floating point computations.

3.1 Magnification Filters

This section is based on the papers *Hardware-Accelerated High-Quality Filtering on PC Hardware* [36], and *Fast and Flexible High-Quality Texture Filtering With Tiled High-Resolution Filters* [39], as well as the technical sketch *Hardware-Accelerated High-Quality Filtering of Solid Textures* [37] presented at SIGGRAPH 2001.

As described in section 2.3, in the case of *texture magnification*, the texture re-sampling filter is dominated by the reconstruction filter, and good results can be achieved without any pre-filter at all. Hence most texture filtering implementations, including graphics hardware, apply only a reconstruction filter when a texture is magnified, i.e., loosely speaking, when a single texel maps to multiple pixels. Therefore, we will call a reconstruction filter used for purposes of texture magnification also a *magnification filter*.

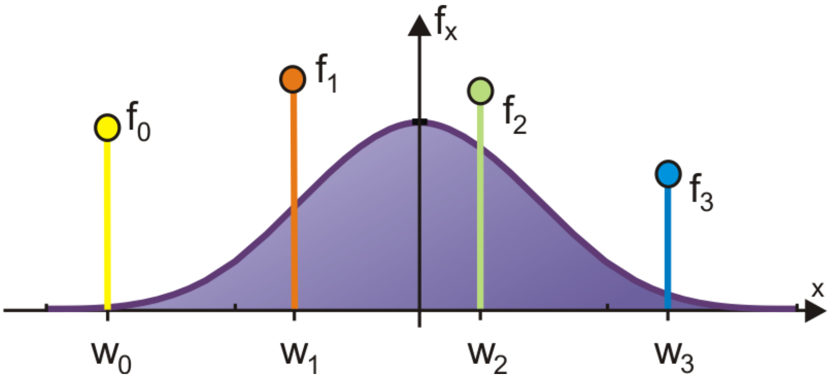


Figure 3.2: A 1D cubic filter kernel subtends four input samples (f_i). The four weights (w_i) needed to evaluate the filter convolution sum depend on the actual re-sampling position x.

A very important property of a magnification filter is that it can be evaluated with constant cost, which is solely determined by the type and size of the filter kernel. To illustrate this, figure 3.2 shows an example of a 1D cubic magnification filter that always uses exactly four input samples in order to calculate a single filtered output sample.

Convolution filtering

Our method achieves high-quality texture filtering by performing a convolution of a discrete input texture with a "continuous" filter kernel, which is pre-sampled and also stored in multiple texture maps.

Fundamentally, all the algorithms presented in this section evaluate the general filter convolution sum, which can be stated for the one-dimensional case as:

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - m+1}^{\lfloor x \rfloor + m} f_i \cdot h(x-i) = \sum_{i=\lfloor x \rfloor - m+1}^{\lfloor x \rfloor + m} f_i \cdot w_i(x) \tag{3.1}$$

where $g(x)$ is the output at re-sampling position x, f_i is the discrete input texture, $h(x)$ is the continuous filter kernel, $m = n/2$ is half the filter width when n is the order (cubic: $n = 4$), and the $w_i(x)$ are the n filter weights corresponding to x. See figure 3.2 for an illustration of convolution with a cubic filter kernel. Examples for filter kernels $h(x)$ of width four are shown in figure 3.3.

The remainder of this section illustrates the basic principle and several algorithms for evaluating equation 3.1 entirely in graphics hardware, by substituting $h(x)$ with a sampled "high-resolution" representation that is stored in several texture maps of user-specified size.

The general method we present exploits multi-texturing and multiple rendering passes for the actual computation. However, advanced features of newer graphics hardware can be used in order to reduce the number of rendering passes and exploit different properties of the filter kernel. On current graphics hardware such as the ATI Radeon 9800, cubic filtering with arbitrary kernels is possible in a single rendering pass as explained in detail below.

Since the filter function $h(x)$ is represented by an array of sampled values, the basic algorithm works irrespective of the shape of this function. However, kernel properties such as

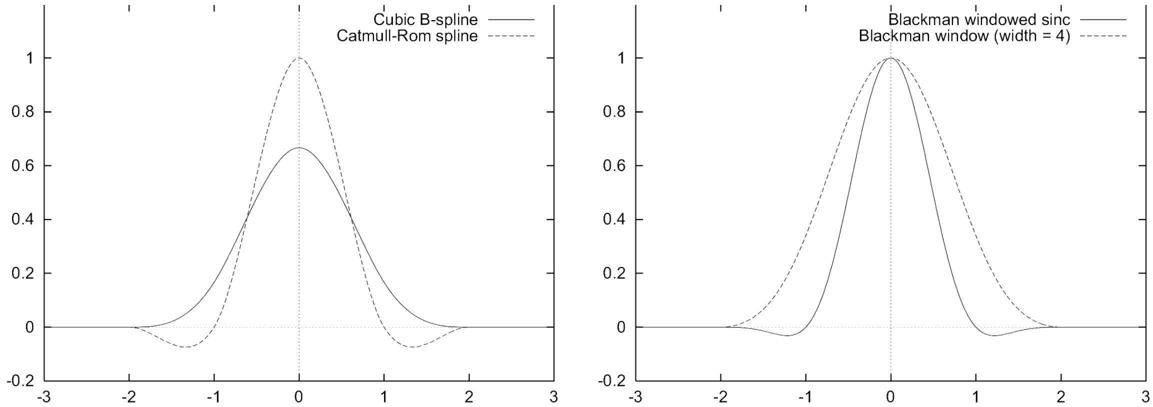


Figure 3.3: Example filter kernels of width four. (left) cubic B-spline and Catmull-Rom spline; (right) Blackman windowed sinc, depicting also the window itself.

separability and symmetry can be exploited to gain higher performance. The basic algorithm is also independent from the dimensionality of input textures and filter kernels. Thus, in the context of texture-based volume rendering, it can be used in conjunction with all kinds of proxy geometry, regardless of whether 2D or 3D textures are used.

Filter kernel representation: tiled high-resolution filters

As shown below, our filtering framework requires the filter kernel to be split up into individual parts of unit extent. In our framework, these parts are called *filter tiles*, and the corresponding textures are referred to as *tile textures*.

In contrast to filters that are used for image processing purposes, this sampled filter kernel representation is of relatively high resolution, in the sense that it contains a much higher number of samples than the width of the kernel itself. Therefore, we also call this representation a *tiled high-resolution filter*.

The exact number of tile textures depends on properties of the filter kernel and the actual convolution algorithm used. Kernels of higher dimensionality than 1D are handled depending on whether they are separable or not. If a kernel is separable, a possible optimization is to store lower-dimensional (usually 1D) kernel tiles, and multiply them on-the-fly at run time, preserving both texture memory and texture cache usage. If the kernel is not separable, it has to be sampled into several texture maps of according dimensionality, i.e., is required to reside in actual 2D or 3D texture maps, respectively.

For example, a general bi-cubic kernel requires sixteen 2D texture maps, which are illustrated in figure 3.4, and a general tri-cubic kernel requires 64 (4^3) 3D textures.

If a filter kernel is symmetric, this property can be exploited as well in order to reduce the number of kernel textures required. A symmetric bi-cubic kernel can be stored in three instead of sixteen textures, for example, and a symmetric tri-cubic kernel can be stored in four instead of 64 textures.

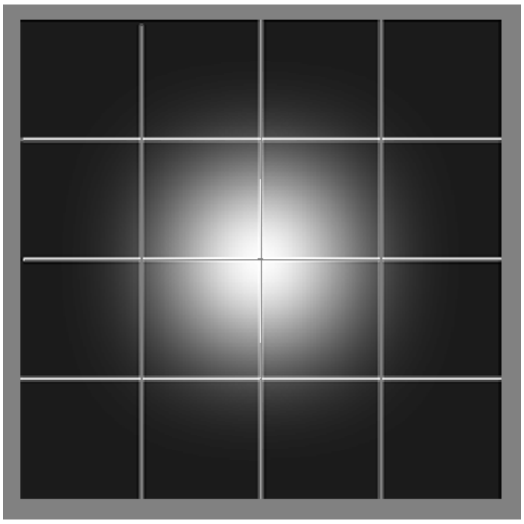


Figure 3.4: 2D filter kernel tile textures of a bi-cubic B-spline. There are sixteen tile textures, each corresponding to a unit square of the 4x4 filter kernel domain.

Fortunately, many important filter kernels – especially many filters used for function reconstruction – are both separable and symmetric. Thus, it is possible to attain both bicubic and tri-cubic reconstruction with only two 1D textures for storing the filter kernel itself. Judging from our experience, a sampling resolution of 64 or 128 samples per filter tile and dimension usually suffices for high-quality reconstruction.

Multi-pass convolution filtering

In the following, we describe a general method for performing on-the-fly filtering with arbitrary convolution filters for texture magnification in multiple rendering passes. This method allows to convolve arbitrary 1D, 2D, or 3D input textures with pre-sampled filter kernels in real-time on a wide range of graphics hardware.

The filter convolution sum is evaluated over multiple rendering passes, in each pass simultaneously point-sampling the input texture, generating the needed filter weights, and multiplying the corresponding input data and weights. The number of rendering passes depends on the width of the filter kernel, the actual convolution algorithm employed, and the maximum number of texture units supported by the hardware. It may be as low as a single pass, and as high as the number of filter tiles.

Generation of filter weights in each pass ranges from simply sampling one of the tile textures, to compositing two or three values or vectors, retrieved from different filter tiles, in the pixel shader. Basically, both the input texture and from one to three tile textures need to be mapped to the output sample grid in screen space multiple times. This mapping is the same for the input texture and the filter tiles apart from scale, where an entire filter tile is mapped to a single input texel. Also, tile textures are replicated automatically, so that the same filter tile maps to every input texel.

Perspective correction is not only used for input textures, but also for tile textures. Thus, all algorithms presented in this chapter are independent from the type of projection employed. They work equally well for both orthogonal and perspective projections.

An important property of these algorithms is that pre-processing of the input data to be filtered is not required for all but one proposed variant. However, we also describe an

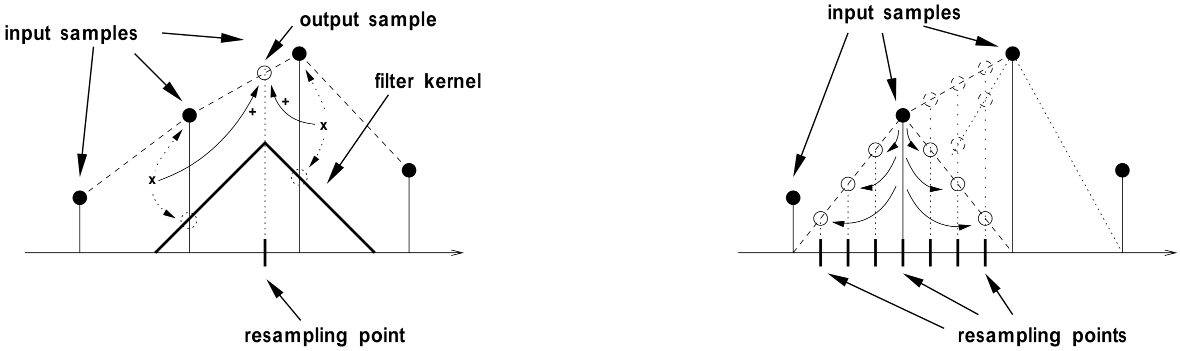


Figure 3.5: Gathering vs. distribution of input sample contributions. (left) gathering all contributions to a single output sample; (right) distributing a single input sample's contribution. Although this example uses a tent filter, the concept is independent from the actual filter.

```
(a) FOR ALL output samples $x_i$ DO
        FOR ALL contributing relative input locations $r_j$ DO
            $g(x_i) \mathrel{+}= f[\mathit{trunc}(x_i) + r_j] * h(\mathit{frac}(x_i) - r_j);$

(b) FOR ALL contributing relative input locations $r_j$ DO
        PAR ALL output samples $x_i$ DO
            $g(x_i) \mathrel{+}= \mathit{shift}_j(f)[\mathit{trunc}(x_i)] * h_j(\mathit{frac}(x_i));$
```

Table 3.1: Evaluating equation 3.1 in the usual order (a), versus the one we are using (b). Each iteration of the outer loop in (b) basically corresponds to a single rendering pass; $r_j \in [-\lceil m \rceil + 1, \lceil m \rceil]$. *PAR* denotes a parallel *FOR* loop.

algorithm operating on monochrome input data pre-interleaved into RGBA data, which can be faster than using unprocessed input data.

Gathering vs. distributing sample contributions

The basic idea of convolution filtering in multiple rendering passes is to use a different evaluation order of the filter convolution sum than the one usually employed in CPU-based filtering.

Instead of computing each output sample $g(x)$ at a point x in its entirety by *gathering* all relevant input sample contributions, we instead *distribute* the contribution of a single relative input sample to all relevant output samples in a single rendering pass.

That is, equation 3.1 would usually be evaluated with code roughly equivalent to what is shown in table 3.1(a): looping over all output locations x_i one after the other, the corresponding output sample $g(x_i)$ is generated by adding up the contributions of the $2m$ contributing neighbor samples, whose locations are specified relative (r_j) to the current output sample location. That is, all the contributions of neighboring input samples (their values multiplied

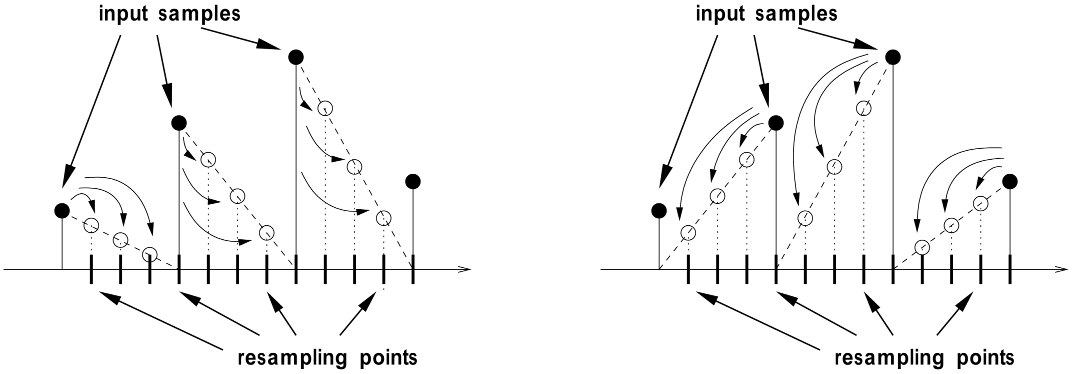


Figure 3.6: Distributing the contributions of all "left-hand" (left), and all "right-hand" (right), respectively, input sample neighbors to the same grid of output samples. This illustration uses a tent filter as an example for illustrating a general concept.

by the corresponding filter values) are gathered and added up in order to calculate the final value of a certain output sample. This gathering of contributions is illustrated in figure 3.5(a). This illustration uses a simple tent filter as an example. It shows how a single output sample is calculated by adding up two contributions. The first contribution is gathered from the neighboring input sample on the left-hand side, and the second one is gathered from the input sample on the right-hand side. For generating the desired output image in its entirety, this is done for all corresponding output sample locations (re-sampling points).

Instead of this usual approach, using high-resolution filter tiles and multiplying input samples by filter weights in the pixel shader amounts to what is shown in table 3.1(b): looping over all relative input sample locations contributing to each respective output sample location, a single rendering pass adds the corresponding contribution of each relative input sample to all output samples simultaneously. That is, the contribution of an input sample is distributed to its neighboring output samples, instead of the other way around. This distribution of contributions is illustrated in figure 3.5(b). In this case, the final value of a single output sample is only available when all corresponding contributions of input samples have been distributed to it. The shift operator $\mathit{shift}_j(\cdot)$ in table 3.1(b) denotes that the input texture is not actually indexed differently at each output sample location, but instead the entire input texture is shifted according to the current rendering pass, in order to retrieve a single *relative input sample location* for all output samples simultaneously. The term relative input sample location denotes a relative offset of an input sample with respect to a set of output samples. Similarly, $h_j(\cdot)$ denotes the filter tile corresponding to the current pass j, instead of the entire filter kernel $h(\cdot)$.

The reason for evaluating the convolution sum in this particular order is that the distribution of the contributions of a single relative input sample can be done in hardware for all output samples (pixels) simultaneously in a single rendering pass using just two textures. The final result is then gradually built up over multiple rendering passes.

Multi-pass convolution by example

We now illustrate multi-pass convolution using the trivial example of a one-dimensional tent filter, which is illustrated in figures 3.6 and 3.7. However, this example is only used for illustration purposes, and it is important to bear in mind that our method is independent of both filter shape and size. Another example, using a Catmull-Rom spline as reconstruction filter, is illustrated in figure 3.8.

In the example of a one-dimensional tent filter, there are two relative input sample locations. One could be called the "left-hand neighbor," the other the "right-hand neighbor." In the first pass, the contributions of all respective left-hand neighbors are computed. The second pass then adds the contributions of all right-hand neighbors. Note that the number of passes depends on the filter kernel, e.g., four instead of two in the case of the Catmull-Rom spline illustrated in figure 3.8.

Thus, the same part of the filter convolution sum is added to the previous result for each pixel at the same time, yielding the final result after all parts have been added up. From this point of view, the graph in figure 3.5(b) depicts both rendering passes that are necessary for reconstruction with a one-dimensional tent filter, but only with respect to the contribution of a single input sample. The contributions distributed simultaneously in a single pass are depicted in figure 3.6, respectively. In the first pass, the contributions of all relative left-hand neighbors are distributed. Consequently, the second pass distributes the contributions of all relative right-hand neighbors. Adding up the distributed contributions of these two passes yields the final result for *all* re-sampling points (i.e., linearly interpolated output values in this example).

Figure 3.7 illustrates this algorithm with multi-texturing of two simultaneous textures. Each pass uses two textures at the same time, the first texture unit point-sampling the original input texture, and the second one sampling the current filter tile texture. These two textures are superimposed, multiplied, and added to the contents of the frame buffer. In this way, the contribution of a single given filter tile to all output samples is computed in a single rendering pass. The input samples used in a single pass correspond to a specific relative input sample location or offset with regard to the output sample locations. That is, in one pass the input samples with relative offset zero are used for *all* output samples, then the samples with offset one in the next pass, and so on. Thus, the number of passes necessary is equal to the number of filter tiles the filter kernel consists of.

Note that the subdivision of the filter kernel into its tiles is crucial to hardware-accelerated convolution filtering. It is necessary in order to attain a correct mapping between locations in the input data and the filter kernel, and to achieve a consistent evaluation order of passes everywhere.

The only basic requirement: 2x multi-texturing

A convolution sum can be evaluated in the way outlined above because it requires only two basic inputs: the set of input samples, and the filter kernel. Because we change only the order of summation but leave the multiplication untouched, we need these two available at the same time. Therefore, we employ multi-texturing with (at least) two textures and retrieve input

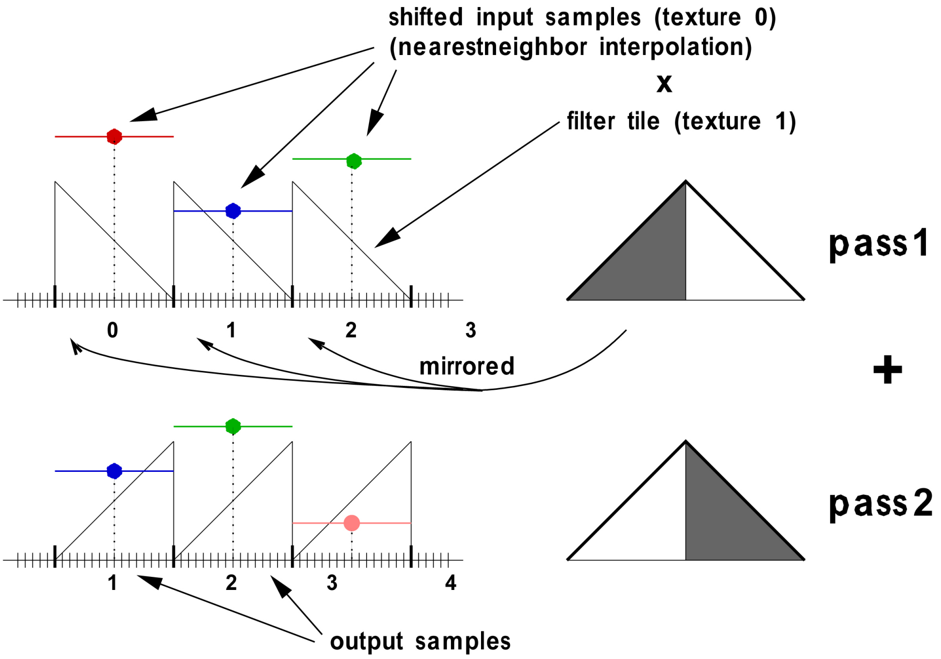


Figure 3.7: Tent filter (width two) used for reconstruction of a one-dimensional function in two passes. Imagine the values of the output samples added together from top to bottom.

samples from the first texture, and filter kernel values from the second texture. Due to the fact that only a single filter tile is needed during a single rendering pass, all tiles are stored and downloaded to the graphics hardware as separate textures.

The required replication of tiles over the output sample grid is easily achieved by configuring the hardware to automatically extend the texture domain beyond $[0, 1]$ by simply repeating the texture via a clamp mode of `GL_REPEAT`. In order to fetch input samples in unmodified form, nearest-neighbor interpolation has to be used for the input texture.

If a given hardware architecture is able to support $2n$ textures at the same time, the number of passes can be reduced by n. That is, with two-texture multi-texturing four passes

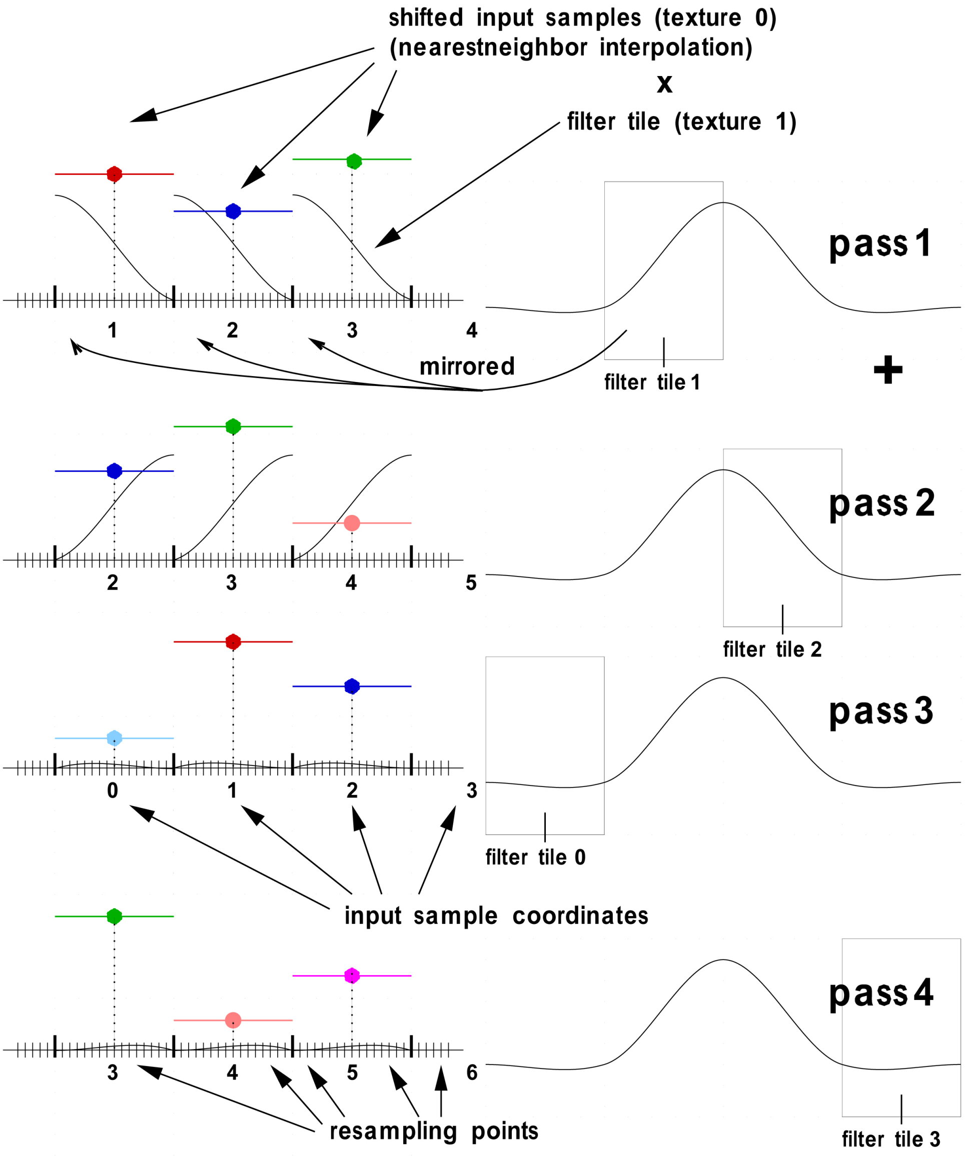


Figure 3.8: Catmull-Rom spline (width four) for one-dimensional function reconstruction in four passes. Imagine the values of the output samples added together from top to bottom.

are needed for filtering with a cubic kernel in one dimension, whereas with four-texture multi-texturing only two passes are needed, etc.

Note that the method outlined above is not considering area-averaging filters, since we are assuming that magnification is desired instead of minification. This is in the vein of graphics hardware using bi-linear interpolation for magnification, and other approaches, usually MIP-mapping, to deal with minification. The extension of our method to the case of minification with MIP-mapping is described in section 3.2.

Convolution algorithms

This section presents a family of actual algorithms for performing convolution with high-resolution filters in hardware. These algorithms are categorized according to whether they assume the filter kernel to be separable, symmetric, or separable and symmetric; and whether they use unmodified RGBA input textures, or require monochrome textures that have been pre-interleaved into RGBA textures.

Each of the basic algorithms can be expanded to make use of more available texture units (and thus fewer passes), by simply adding the results of multiple basic blocks together in a single pass. Combining multiple logical rendering passes also helps to exploit higher internal calculation precision in order to reduce quantization artifacts if 8-bit operations instead of floating point computations are used.

We illustrate the differences with pseudo code fragments showing the global setup (for all passes), setup that changes per pass, and the vertex and pixel shaders executed in each pass. For the actual implementations of the latter two, we use either the `NV_vertex_program` and `NV_register_combiners` (NVIDIA GeForce 3+), or the `EXT_vertex_shader` and `ATI_fragment_shader` (ATI Radeon 8500+) OpenGL extensions.

General filter kernels

In the general case, we assume that the filter kernel is used and stored in its entirety, i.e., potential separability or symmetry is not exploited. This implies that all tile textures have the dimensionality of the convolution itself (1D, 2D, or 3D). Naturally, this case has the highest demands on texture memory and texture cache usage. Table 3.2 shows the pseudo code for the resulting algorithm, which we call *std-2x*. “std” meaning the standard algorithm (not separated, non-interleaved), and “2x” denoting the number of textures used in a single pass (in this case two). Several of these building blocks can be combined in a single rendering pass, depending on the number of texture units supported by the hardware, thus yielding the analogous extended algorithms *std-4x*, *std-6x*, and so on. The basic algorithm works identically for 1D, 2D, and 3D convolutions, respectively, apart from minor details such as the number of texture coordinates that need to be configured.

Symmetric filter kernels

Exploiting filter kernel symmetry can easily be done be reusing tiles with mirrored texture coordinates. Symmetry can be exploited both along axes and diagonals. Especially when the filter is not separable, it is important to exploit symmetry, in order to reduce texture memory consumption, which is especially high in the case of 3D convolutions.

It is important to note here that mirroring filter tiles is not necessarily as simple as swapping texture coordinates. The reason for this is that the mirroring is required to be pixel-exact. That is, if a specific pixel in screen space was covered by a specific weight in a filter tile in one pass, it must be covered by the exactly corresponding weight in the respective mirrored tile in another pass. While it might simply suffice to mirror texture coordinates by, e.g., negation in the vertex shader, on some hardware architectures it may be necessary to actually mirror tiles in the pixel shader. Although texture coordinate iteration can be exact enough for coordinates to be mirrored only at the vertices and still achieve consistent results in different passes, the only way to guarantee pixel-exactness is mirroring on a per-pixel basis.

The algorithm outlined in table 3.2 contains an "if kernel symmetric" clause, which denotes where tile mirroring needs to be setup in order to exploit symmetric filter kernels.

Separable filter kernels

When the filter kernel is separable, the texture memory requirements and cache usage can be reduced tremendously. Instead of using actual 2D or 3D tile textures for 2D or 3D convolutions, the higher-dimensional weights can be generated on-the-fly from either two or three one-dimensional tile textures in the pixel shader. This is easily possible by performing separable composition, i.e., multiplying corresponding weights retrieved from two – not necessarily different – lower-dimensional tiles. Alternatively, for 3D convolutions the needed filter weights can also be generated from one 1D texture and one 2D texture, e.g., in order to lower the number of texture units required.

Table 3.3 shows the corresponding algorithms. The *sep-3x* algorithm can be used for 2D or 3D convolution. In the former, two-dimensional filter tiles are substituted by two one-dimensional tiles. In the latter, one 1D tile and one 2D tile are used instead of a single three-dimensional tile.

Although in theory there would not be any difference between exploiting separability by generating filter weights on-the-fly, and not utilizing it by simply performing separable composition beforehand when building tile textures, in practice the results are slightly different. The reason for this is that in a pre-process the necessary multiplications can be carried out

```
global setup:
    input texture:   as-is, all formats (mono, RGB, RGBA, etc.);
    filter tiles:   1D, or not-separated 2D or 3D;
per-pass setup:
    filter tiles:   select tile corresponding to pass;
    if kernel symmetric:   setup tile mirroring;
vertex shader:
    texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
    texcoord[ UNIT1 ].s{t{r}} = texcoord[ UNIT0 ] * tile_size;
pixel shader:
    reg0 = SampleTex( UNIT0 );
    reg1 = SampleTex( UNIT1 );
    out = Multiply( reg0, reg1 );
```

Table 3.2: Convolution with *std-2x* algorithm.

in floating-point precision, whereas in the pixel shader they have to be done at whatever precision the hardware offers there. Although rarely noticeable, exploiting separability or not is definitely a quality/performance trade-off. While we have also used numerical simulations for estimating the numerical difference, the best way to prefer one algorithm over the other is by visual comparison.

Separable and symmetric filter kernels

The best possible combination of kernel properties is when a filter is both separable and symmetric, which fortunately many interesting filter kernels are – especially many of those one would like to use for function reconstruction purposes. The pseudo code in table 3.3 contains an “if kernel symmetric” clause, which denotes where kernel symmetry needs to be taken into account. Apart from this, the algorithm is identical to the separable-only case.

Pre-interleaved monochrome input

If the input data is single-valued and a certain limited amount of pre-processing is considered feasible, all of the algorithms outlined above can be combined with the following scheme that exploits the capability of graphics hardware to perform per-pixel dot products.

The idea is to fold four passes into a single pass, by evaluating four terms of the convolution sum in a single operation. In order to do so, four input values have to be available simultaneously, which can be achieved by converting a monochrome input texture into an RGBA texture by interleaving it four times with itself, each time using a different texel off-

```
global setup:
  input texture:  as-is, all formats (mono, RGB, RGBA, etc.);
  filter tiles:  separable; only 1D (3x/4x), or 1D plus 2D (3x);
per-pass setup:
  filter tiles:  select tiles corresponding to pass;
  if kernel symmetric:  setup tiles mirroring;
vertex shader:
  texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
  texcoord[ UNIT1 ].s{t} = texcoord[ UNIT0 ] * tile_size;
  texcoord[ UNIT2 ].s = texcoord[ UNIT0 ] * tile_size;
  texcoord[ UNIT3 ].s = texcoord[ UNIT0 ] * tile_size;  // sep-4x
pixel shader:
  reg0 = SampleTex( UNIT0 );
  reg1 = SampleTex( UNIT1 );
  reg2 = SampleTex( UNIT2 );
  reg3 = SampleTex( UNIT3 );  // sep-4x
  reg2 = Multiply( reg2, reg3 );  // sep-4x
  reg1 = Multiply( reg1, reg2 );
  out = Multiply( reg0, reg1 );
```

Table 3.3: Convolution with *sep-3x* and *sep-4x* algorithms; only the latter contains the statements marked with *// sep-4x*.

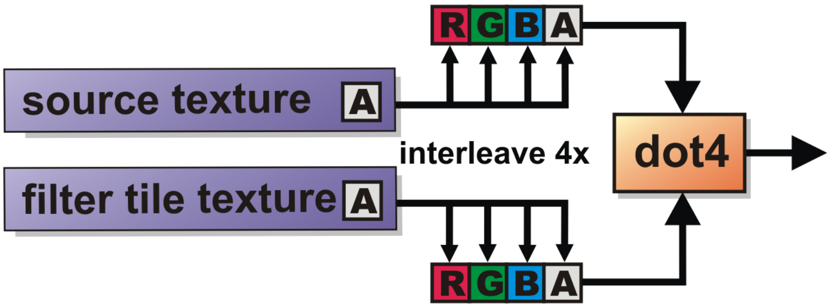


Figure 3.9: Interleaving both the source and filter tile textures four times allows to fold four passes of evaluating the convolution into a single pass by using a dot-product operation instead of simple multiplication.

set. That is, at each position in the texture the value of three neighboring texels is available in addition to the original texel itself. If the tile textures are also interleaved accordingly, thus achieving correct correspondences of relative input samples and filter weights in all four channels, four terms of the convolution sum can then be evaluated concurrently by using a four-component dot product instead of a component-wise multiplication in the pixel shader. This approach is illustrated in figure 3.9.

The *dot-2x* pseudo code in table 3.4 illustrates this as an extension of the *std-2x* algorithm of table 3.2. Although not shown here for brevity, the same approach can also be combined trivially with the separable algorithms outlined in table 3.3, thus yielding the *spd-3x*, *spd-4x*, *spd-6x*, etc. algorithms.

Exploiting both separability and pre-interleaved input data is generally the fastest approach offered by our framework. Note that the interleaved source texture is of exactly four times the size of the corresponding monochrome texture. Or, comparing with the size of an RGBA input, the size stays the same, but color information is lost.

```
global setup:
    input texture:  monochrome, but interleaved in RGBA;
    filter tiles:  1D, 2D, or 3D; analogously interleaved in RGBA;
per-pass setup:
    filter tiles:  select tile corresponding to pass;
    if kernel symmetric:  setup tiles mirroring;
vertex shader:
    texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
    texcoord[ UNIT1 ].s{t{r}} = texcoord[ UNIT0 ] * tile_size;
pixel shader:
    reg0 = SampleTex( UNIT0 );
    reg1 = SampleTex( UNIT1 );
    out = DotProduct4( reg0, reg1 );
```

Table 3.4: Convolution with *dot-2x* algorithm.

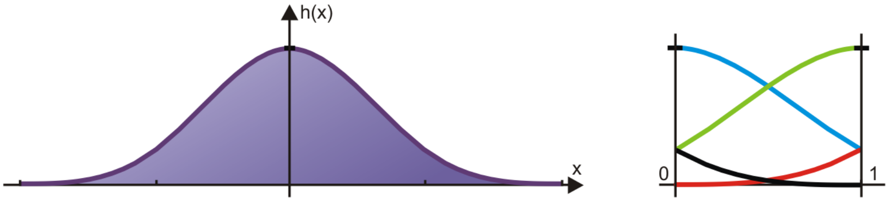


Figure 3.10: A 1D cubic filter kernel (here: cubic B-spline) is sampled and stored in the four channels of a single RGBA texture map. A single texture fetch operation retrieves the four weights needed for the evaluation of the filter convolution sum at any re-sampling location.

Pre-interleaved filter tile textures

A very interesting alternative to the interleaved approach outlined above is to avoid interleaving of the source texture, but use interleaved filter tile textures. This requires flexibility in the pixel shader in order to route filter weights to the corresponding input samples, but on current hardware architectures we consider this algorithm to offer the best trade-off of all variants described in this section.

This approach is especially attractive for filtering with separable cubic filter kernels, since in this case all the necessary kernel information fits into a *single* 1D kernel tile texture. Figure 3.10 shows a cubic B-spline converted into a single tile texture by storing each of its four filter tiles into the R, G, B, and A channels of a single RGBA texture, respectively.

Replicating this texture over the entire output sample grid allows to fetch all four filter weights necessary for evaluating the convolution via a single texture fetch, see figure 3.11. The cases of bi-cubic and tri-cubic filtering can be handled analogously, by stretching this single 1D texture two or three times, respectively, over the output sample grid and multiplying corresponding filter weights in the pixel shader.

Interpolation by pre-processing

Using filter kernels that are non-negative over their entire domain simplifies handling the corresponding tile textures and evaluating the filter convolution sum. However, such kernels are not able to perform interpolation of the original data.

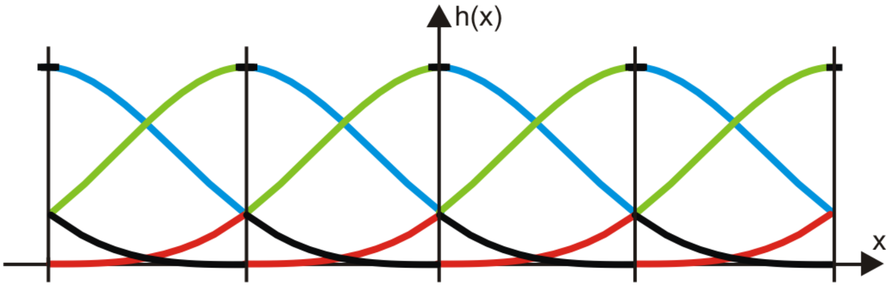


Figure 3.11: Replicating an interleaved RGBA filter tile texture over the output sample grid yields a distribution of basis functions familiar from spline curves. All four weights needed for convolution at any given re-sampling location are available via a single texture fetch.

A very interesting approach to avoid negative values in the filter kernel and still be able to achieve interpolation instead of just approximation is to pre-process the input data in such a way that applying a B-spline filter, for example, yields interpolated results [153]. In order to do this, we modify the filter convolution sum (equation 3.1) slightly:

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} c[i]h(x - i) \tag{3.2}$$

The difference here is that we do not filter the actual data points but special coefficients $c[i]$, and require the resulting function to interpolate the original data. The coefficients depend on the actual filter used and can be calculated by matrix inversion [180]. We can now use a non-negative filter (e.g., cubic B-spline) and still get high-quality interpolation results [153].

The coefficients $c[i]$ are calculated in a pre-processing step in software. When 8-bit fixed point computations are used, storing theses coefficients directly in a texture is usually not possibly, since in general they exceed the range $[0, 1]$. In this case, we fit them into this range using simple scale and bias operations to avoid clamping artifacts. After filtering with the non-negative filter, where we render to a texture instead of the frame buffer, one additional rendering pass is required to restore the correct intensity values with an inverse scale and bias operation. Although the last pass is done in the pixel shader, the required render-to-texture operation is rather time-consuming.

3.2 Minification Filters and MIP-Mapping

This section is based on the technical sketch *MIP-Mapping with Procedural and Texture-Based Magnification* [38] presented at SIGGRAPH 2003.

A very common method to deal with texture minification is to use MIP-mapping [178], a concept which is especially important in hardware texture-mapping. Although in some applications, e.g., volume rendering [174], textures are commonly used without MIP-mapping since they are magnified most of the time, many applications have to deal with both magnification and minification of textures. However, higher-order magnification filters cannot be used directly in conjunction with MIP-mapped textures without incurring artifacts in lower-resolution MIP-map levels.

This section shows how the higher-order magnification filters introduced in the previous section can be combined with MIP-mapping in order to be able to use them for both texture minification and magnification, which enables their use in a wide variety of applications. In order to do this, the filter kernel is adapted to the actual MIP-map level used for a given pixel on a per-pixel basis, which allows using arbitrary convolution filters for filtering MIP-mapped textures.

We achieve this by retrieving per-pixel MIP-map level information from a *meta MIP-map*, and adapting the filter kernel accordingly. The bottom two images of figure 3.1 illustrate the difference between cubic magnification of a MIP-mapped texture without and with per-pixel correction of the filter kernel, respectively.

An approach similar to our meta MIP-map has also been used in the context of shadow mapping [26], and estimating screen space derivatives of quantities such as texture coordinates interpolated during triangle rasterization [129].

Dependence on input texture resolution

The framework for convolution filtering introduced in the previous section depends on knowledge of the exact resolution of the input texture, which turns out to be the major obstacle when filtering MIP-mapped textures.

First, the input sample values needed must be made available to the pixel shader. In texturing hardware, fast direct access to input samples is not possible, but the same effect can be achieved by using the input texture together with nearest-neighbor interpolation as reconstruction filter. In this way, unaltered input sample values can be made available to the pixel shader. In order to evaluate the convolution sum, the input texture is mapped to the same pixel multiple times with different offsets of whole texels in order to make all input samples to the convolution sum accessible.

In order to be able to do this, the input texture resolution must be known in order to create offsets of whole texels in the texture coordinate domain. Since the entire texture corresponds to a coordinate domain of $[0, 1]$ [1], a single texel is of size $1/texture\_size$. For example, in a texture of size 64, a single texel has size 1/64, etc. Naturally, for two-dimensional or three-dimensional textures, the width, height, and depth of a texel in the texture coordinate domain can be different and usually need to be calculated separately.

Second, the size must also be known in order to calculate the correct filter weights (in procedural convolution [4]), or retrieve filter weights from the correct locations of a filter kernel texture (in texture-based convolution). The fundamental operation in this case is a multiplication of the input texture coordinates by the size of the input texture, i.e., a scale factor of 64 for an input texture of size 64.

When MIP-mapping is not used, the input texture resolution can simply be made available to the pixel shader as a constant parameter. In order to avoid unnecessary computations in the pixel shader, the size of an individual texel in the texture coordinate domain and the scale factor for filter weight generation or sampling can be specified instead. Texture coordinates can also be generated entirely in the vertex shader, which reduces the load on the pixel shader.

Combination with MIP-mapping

In the presence of MIP-mapping, the problem of providing the pixel shader with the correct input resolution information becomes non-trivial.

The input grid resolution is changed by the hardware on a per-pixel basis by automatic selection of a suitable MIP-map level, and this process is entirely transparent to the pixel shader. That is, even though the pixel shader explicitly requests a given texture to be sampled, it does not know the actual resolution of the texture image that has been sampled if the texture is MIP-mapped. This leads to the problem that correct matching of input samples and filter weights in order to evaluate the filter convolution sum is impossible without knowing the MIP-map level that is actually used for sampling a given texture. To complicate matters further, this MIP-map level cannot even be queried by the pixel shader.

The next section takes a closer look at the problem that occurs when the filter kernel is scaled with respect to the base level image of a MIP-map, which is the natural choice if

---

[1] We assume power-of-two textures with texture coordinates of $[0, 1]$ mapping to an entire texture, instead of rectangular textures where $[0, 1]$ maps to a single texel. Current rectangular texture implementations do not support MIP-mapping at all and are also not available on all platforms.

this scaling has to be constant. Section 3.2 then introduces the concept of a *meta MIP-map*, which we use in order to solve this problem on a per-pixel basis.

MIP-mapping and higher-order filtering with constant filter kernel size

If compensation for the actual MIP-map level used for a given pixel is not done on a per-pixel basis, which we propose in the next section, the filter kernel is usually scaled with respect to the base level. In that case, the higher-order magnification filter gradually degenerates to nearest-neighbor interpolation for lower-resolution MIP-map levels. Figure 3.12 illustrates this for three MIP-map levels of lower resolution than the base level.

If the actual MIP-map level resolution is half the resolution assumed by the filter kernel evaluation, adjacent input samples will effectively be replicated once each (in one dimension; figure 3.12, top row). This replication is a side effect of the way an array indexing operation is simulated in graphics hardware. Since direct access to a given sample in a texture is not possible, the texture must be sampled using nearest-neighbor interpolation instead. However, if the assumed spacing in the texture coordinate domain between two adjacent samples is not correct, nearest-neighbor interpolation will retrieve the same sample multiple times.

Depending on the resampling location, a cubic filter will only use two or three different input samples instead of four and thus not be able to achieve the desired result (figure 3.12, top row). Starting with the next-lower resolution MIP-map level (figure 3.12, second row), resampling areas start to appear where the reconstruction result is a piecewise constant function and thus the filter starts to behave in a way that is similar to nearest-neighbor interpolation. This reduction of reconstruction quality increases rapidly with each MIP-map level (figure 3.12, bottom row). With each successive decrease in resolution, more and more pixels are effectively filtered using nearest-neighbor interpolation. Visually, the result for a resolution mismatch of 4:1 (two MIP-map levels away from the base level) is already almost identical to nearest-neighbor interpolation, which can also be seen in the bottom left image of figure 3.1.

That is, a straightforward combination of MIP-mapping and higher-order filtering results in filtering quality considerably worse than linear interpolation for many MIP-map levels, which is clearly not what we would expect from a cubic filter, for instance. In the next section, we show how to remove the resulting artifacts and attain consistent higher-order filtering of all MIP-map levels by automatically adapting the higher-order filter to the actual MIP-map level used for a given pixel.

The meta MIP-map

Scaling the filter kernel and adjusting the texture coordinates for fetching input samples once the actual input texture resolution is known in the pixel shader is trivial on current graphics hardware. However, on current hardware it is not directly possible to determine the MIP-map level used for a given pixel, and thus the actual per-pixel input texture resolution. For interpolation between adjacent MIP-map levels, even two input texture resolutions have to be considered for a single pixel.

So the problem becomes how to determine information about the actual MIP-map level or levels used by the hardware for any given pixel.

An additional instruction for determining MIP-map information in the pixel shader would be extremely useful. However, all the required information can also be retrieved from an

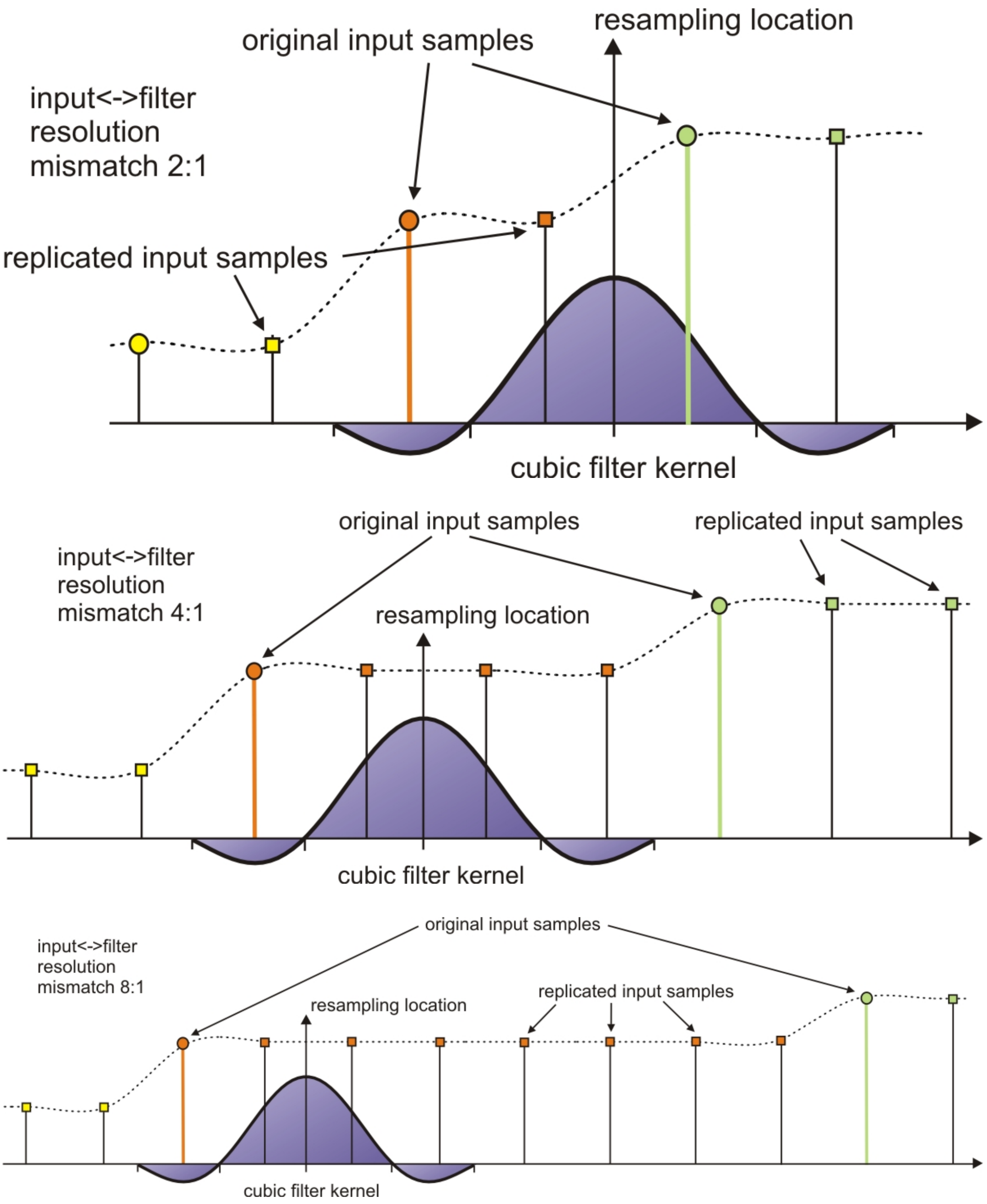


Figure 3.12: The input samples used by the reconstruction filter depend on a proper scaling of the filter kernel with respect to the input texture, which can only be done if the input texture resolution is known. Incorrect scaling leads to replication of input samples with respect to the reconstruction filter. The filter effectively uses fewer and fewer input samples, more and more degenerating to nearest-neighbor interpolation.

additional texture that is MIP-mapped and used identically to the actual image texture. This is possible since two textures that are of the same size and mapped identically will always use exactly the same MIP-map level for any given pixel.

That is, in order to get access to all the required MIP-map level information in the pixel shader, we use an additional MIP-mapped texture that contains no texture data per se, but information about the MIP-map levels themselves. Therefore, we call this texture a *meta MIP-map*.

All entries in a given level of a meta MIP-map contain *identical* values. This guarantees that all pixels using this MIP-map level have access to the corresponding MIP-map meta information.

The meta MIP-map can either contain all the information that we need (MIP-map level, size of a single texel, filter scale, see below) directly, or simply contain lookup values into another texture that in turn contains all the necessary information. The choice depends both on the graphics hardware architecture, and memory overhead considerations (see section A).

Meta MIP-map contents

The meta information that we need includes:

- The s and t texture coordinate scaling factors for matching filter size and MIP-map level, specified relative to the base level to stay within a $[0, 1]$ range (e.g., 0.25 for a MIP-map level of size 16 and a base level of size 64).
- The s and t texture coordinate offsets to get from one texel to the next in a given MIP-map level (e.g., 1/64 for a MIP-map level of size 64).
- The MIP-map level itself, if interpolation between levels is used (i.e., a `GL_*_MIPMAP_LINEAR` minification filter has been specified). If MIP-mapped float textures are not available, the level must be scaled in order to fit into a $[0, 1]$ range, see below.

The meta MIP-map is then used in the pixel shader for adjusting the interpolated texture coordinates to reflect the per-pixel input texture resolution.

Due to precision requirements in texture coordinate arithmetic, both the computations and the (s, t) scale factors and offsets should use floating point precision. This does not necessarily require the meta MIP-map itself to contain floating point data (see below). Floating point values can be stored in an additional 1D texture used as a look-up table, in which case the meta MIP-map itself then only stores fixed point indexes.

Details relevant in practical implementations of the meta MIP-map concept, especially with respect to memory usage and hardware architecture dependencies, are described in Appendix A.

Filtering a single MIP-map level

The resolution of the base level is set as a constant pixel shader parameter, and simple multiplication with the (s, t) texture coordinate scaling factors from the meta MIP-map yields the scaling factors for the filter kernel that guarantee correct calculation (procedural convolution) or retrieval (texture-based convolution) of weights.

The input texture offsets are either specified by the pixel shader itself, or also specified from outside via the same pixel shader constant used to store the base level size. Simple multiplication with the (s, t) texture coordinate offsets from the meta MIP-map yields the correct texture coordinate offsets to retrieve all input samples required by the filter convolution sum.

These two components combined yield a correct `GL_CUBIC_MIPMAP_NEAREST` filter, for example. Since texture-based higher-order filtering can be used with arbitrary filter kernel widths, other higher-order `GL_*_MIPMAP_NEAREST` filters are also possible.

Filtering two adjacent MIP-map levels

If the MIP-map level itself is also stored in the meta MIP-map, it is possible to create `GL_*_MIPMAP_LINEAR` minification filters with higher-order reconstruction. We only need to observe that the fractional part of the result of the linear interpolation between two adjacent meta MIP-map levels containing the MIP-map level itself yields the interpolation weight that has actually been used by the hardware to interpolate between these levels. The pixel shader can then blend two MIP-map levels that have been filtered with a higher-order filter using this weight, and thus attain smooth transitions between MIP-map levels.

Note that in order to be able to filter two adjacent MIP-map levels with a custom filter, the input texture itself must be sampled twice and be filtered with a `GL_NEAREST_MIPMAP_NEAREST` minification filter. In order to get access to two adjacent MIP-map levels, the LOD of both texture fetches must be biased accordingly, e.g., using the `TXB` pixel shader instruction of `ARB_fragment_program` and bias values of 0.5 and -0.5, respectively. The meta MIP-map itself, however, uses `GL_NEAREST_MIPMAP_LINEAR` minification and thus yields the corresponding inter-level interpolation weight.

For storing the MIP-map level itself in the meta MIP-map, it is easiest to use a floating point texture. In this case, floating point textures must support MIP-mapping, which is currently the case on the ATI Radeon 9700, for example, but not on the NVIDIA GeForce FX. However, a scaled version of the MIP-map level fitting into a $[0, 1]$ domain can also be used for obtaining a lower-precision interpolation weight. In this case, the value must be scaled back to the MIP-map level in the pixel shader before taking the fractional part.

Section summary and conclusions

After showing that higher-order texture filtering degenerates to nearest-neighbor interpolation for lower-resolution MIP-map levels yielding worse results than linear interpolation for those levels, a result that one usually would not expect, we have described the concept of a meta MIP-map to solve this problem. In practice, the meta MIP-map concept has proven to be very powerful, and the memory overhead can be much lower than initially expected. However, current pixel shader programming interfaces only partially support enough access to MIP-mapping information. First, it would be very powerful to get access to the MIP-map level used for a given pixel without the need for a meta MIP-map in the first place. Second, full access to the Jacobian matrix is very useful for many applications, in our case reducing the memory overhead of the meta MIP-map concept to a negligible amount. Unfortunately, access to the Jacobian is currently not supported in vendor-independent pixel shader interfaces.

3.3 Surface and Solid Texture Filtering

This section shows examples of using the filtering framework introduced in this chapter for filtering both surface and solid textures.

Surface texture (2D) convolution

The framework presented in this chapter can be used as a high-quality, but still very fast, substitute for the most widely used hardware method for reconstruction of surface textures, e.g., using bi-cubic instead of bi-linear interpolation. We have used cubic B-splines and Catmull-Rom splines, and windowed sinc filters with Kaiser and Blackman windows of width four for this purpose.

Static textures

Figure 3.13 shows a zoom-in of a 64x64 resolution texture mapped onto a 3D surface object, which illustrates that especially for textures of low resolution a higher order filter kernel can make a tremendous difference with respect to reconstruction quality. A case where textures of even much lower relative resolution are used frequently, are light maps for real-time display of radiosity lighting solutions, e.g., in computer games. Also, in the case of dynamic lighting with projective texture mapping [143], the resolution relative to the base texture is often very low, making linear interpolation artifacts strongly visible. Our framework filters in texture space and is therefore independent from the kind of projection used. Textures can be mapped to any underlying geometry.

Another example for a 3D surface object mapped with a 2D texture is depicted in fig-

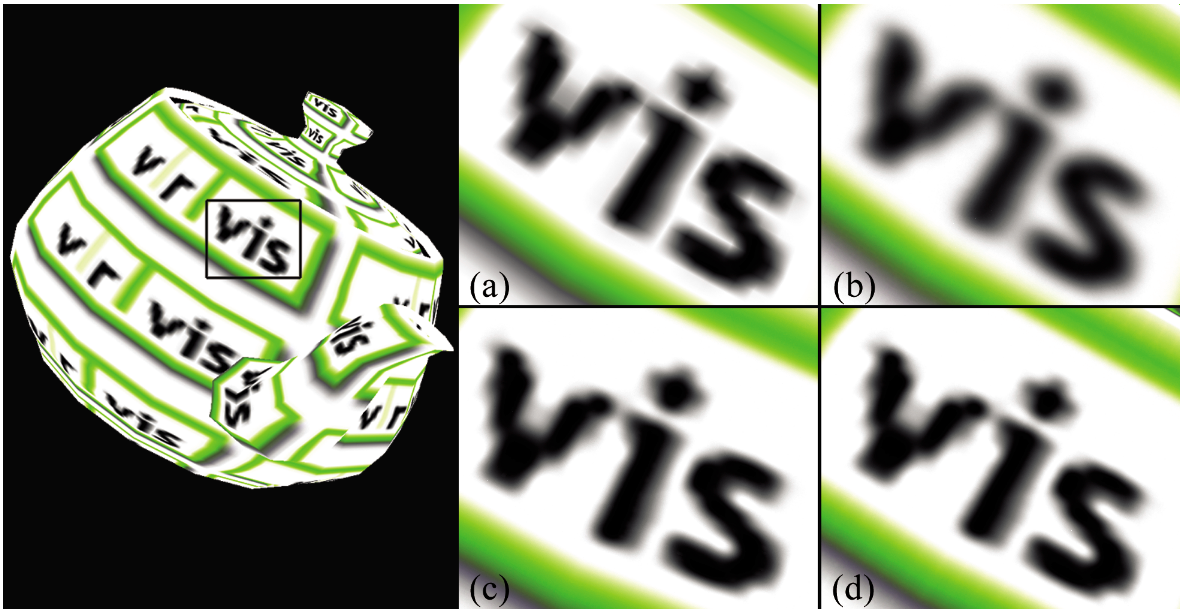


Figure 3.13: Filtering a 64x64 texture mapped several times onto a geometric object; (a) bi-linear interpolation; (b) cubic B-spline; (c) Catmull-Rom spline; (d) Kaiser-windowed sinc.

ure 3.14, where in addition to using high-quality vs. hardware-native reconstruction, a real-time Sobel filter has been used in screen space.

An important constraint to bear in mind is that if convolution is performed in multiple rendering passes, transparent polygons cannot be handled directly. As in all multi-pass algorithms, the computation split up into these passes must not interfere with the blending operation used for transparency. Thus, transparent polygons must first be filtered in an

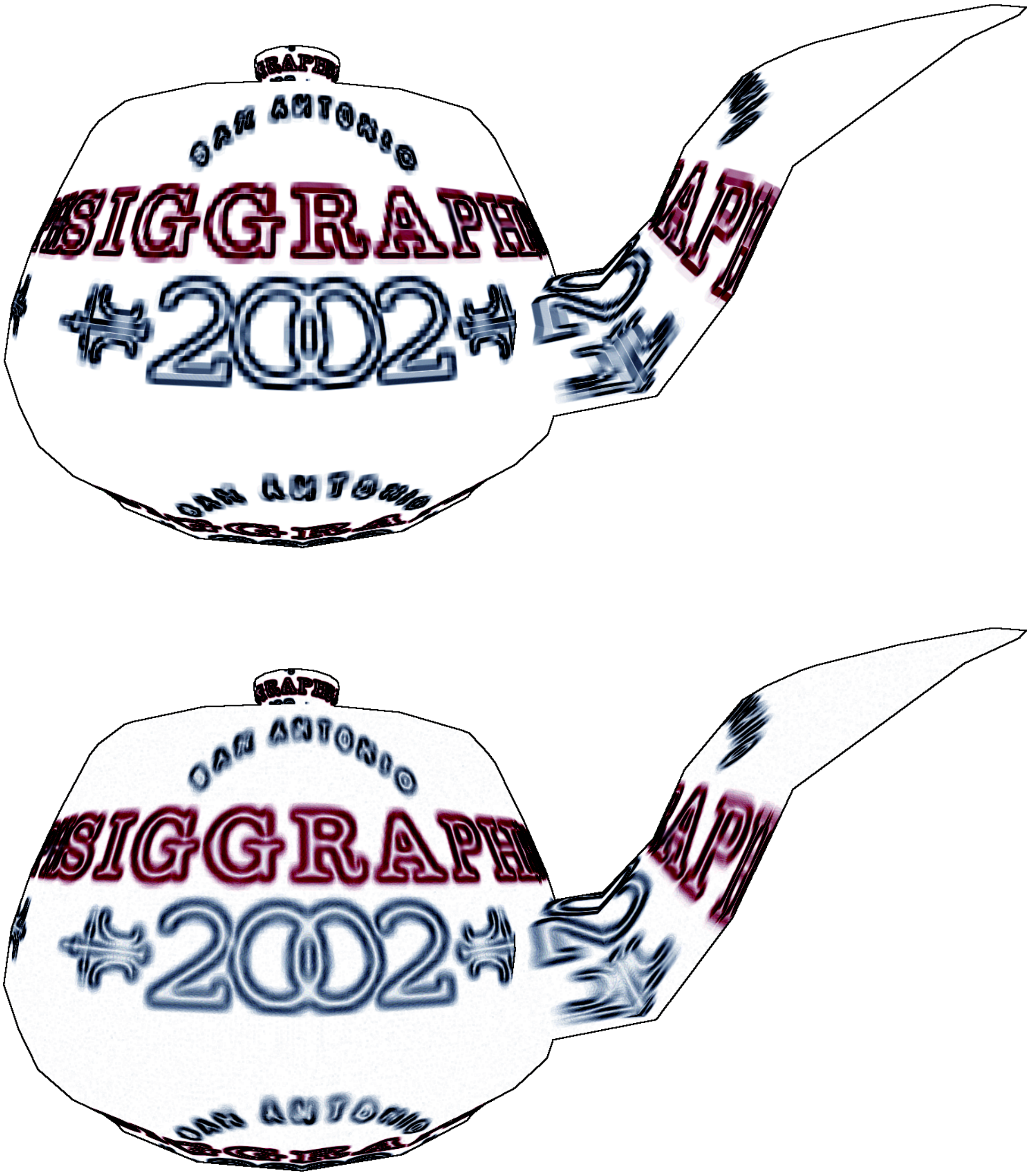


Figure 3.14: Teapot mapped with a 128x64 texture. (top) bi-linear interpolation; (bottom) high-quality reconstruction via 2D convolution with a bi-cubic B-spline; additionally, real-time convolution with a Sobel filter was performed in screen space to enhance edges.

off-screen buffer, and this filtered result must then be used when blending into the frame buffer.

Pre-rendered texture animations

In texture animations, such as the one shown in figure 3.15, the artifacts of linear interpolation are even more pronounced than in the case of static textures. The underlying grid appears as "static" layer beneath the texture itself. These artifacts are successfully removed (to a sufficient extent) by bi-cubic interpolation. Additional examples are shown in figures 3.16 (top row) and 3.18.

Procedural texture animations

When texture animations are generated procedurally, lower texture resolution speeds up the generation process, because fewer samples need to be generated. If the texture is generated using the CPU, this also decreases download time to the graphics hardware. Figure 3.16 (bottom row) shows a comparison of two low-resolution procedural textures generated on the

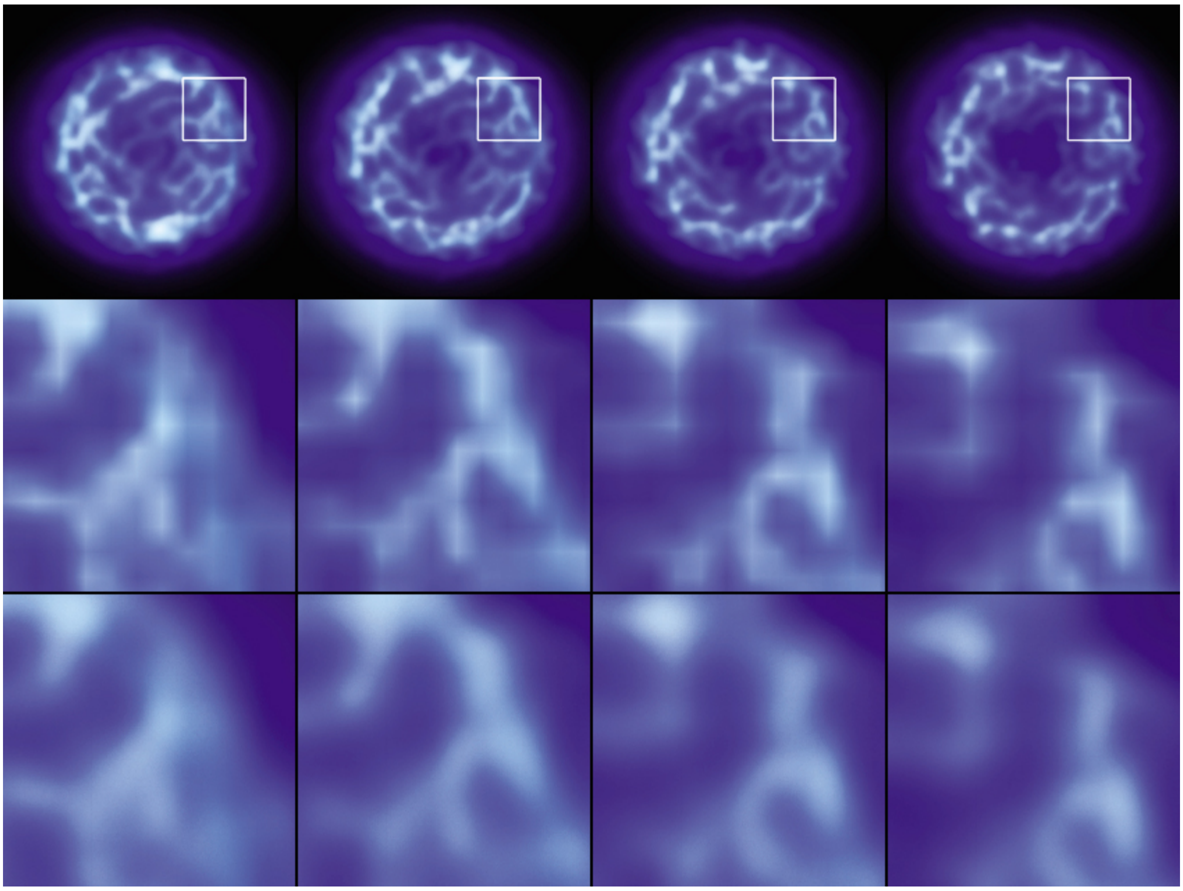


Figure 3.15: Four frames of a 2D particle animation from the space combat game Parsec [122], employing high-resolution convolution for high quality reconstruction; middle row: bi-linear interpolation; bottom row: bi-cubic B-spline.

GPU itself, and subsequently filtered on-the-fly by the hardware.

Solid texture (3D) convolution

All the applications presented above also arise in three dimensions. Although in 3D an increased number of rendering passes and – in the case of a non-separable kernel – more texture memory are needed than in 2D, it is still possible to filter solid or volumetric textures at real-time frame rates.

Static textures

Due to their highly increased memory consumption, solid textures are usually of rather low resolution. Figure 3.17 shows an object mapped with a solid texture, where tri-cubic convolution significantly enhances image quality.

Animated textures

Although several frames of 3D textures usually consume too much texture memory for feasible use, such textures can be generated procedurally on-demand. Especially when the procedural

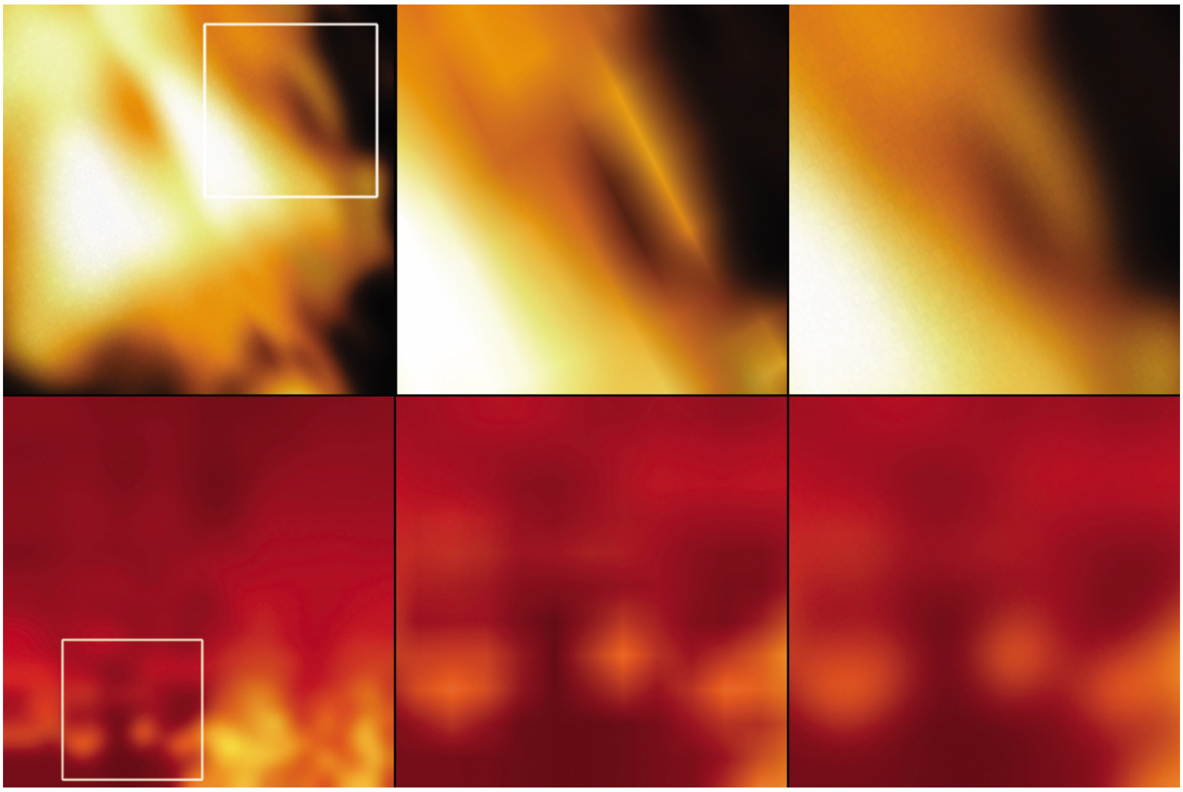


Figure 3.16: Top row images show a frame from a pre-rendered procedural fire animation. Bottom row images show a frame from a procedural fire animation that is generated on-the-fly on the GPU itself. Magnified regions have been filtered with bi-linear interpolation (center) and a bi-cubic B-spline (right).

texture is also generated in hardware, this is feasible for low resolutions. Using higher-order filtering again helps to overcome artifacts from the low-resolution sampling.

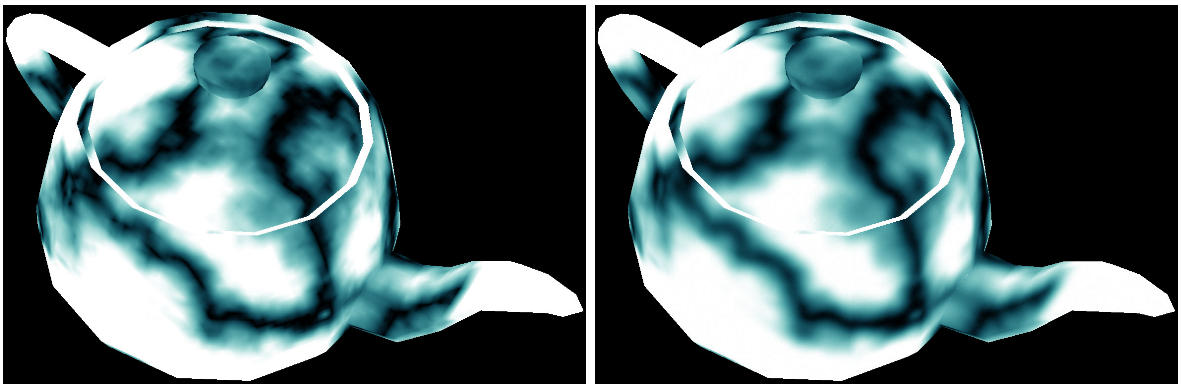


Figure 3.17: Polygonal object mapped with a 128^3 solid texture. (left) tri-linear interpolation; (right) tri-cubic B-spline with real-time 3D convolution. Texture reconstruction is only performed on the actual object surface.

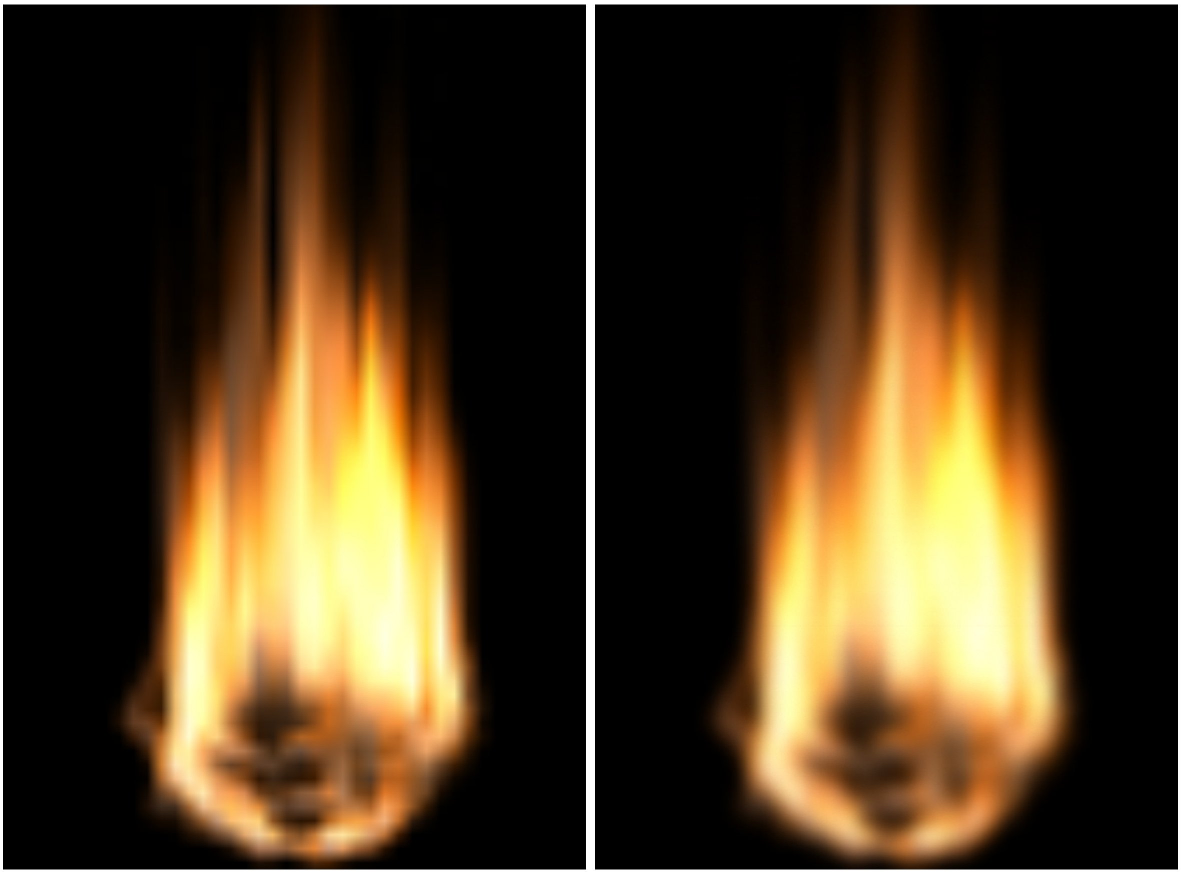


Figure 3.18: Pre-rendered 2D texture animation of a flame. (left) bi-linear interpolation; (right) filtering with bi-cubic B-spline. The left image exhibits many interpolation artifacts especially at the base of the flame, and vertical line artifacts at the top.

3.4 Volume Texture Filtering

This section shows examples of using the filtering framework introduced in this chapter for filtering volume textures for volume rendering.

Reconstructing slices

The basic building block of texture-based volume rendering is rendering individual slices through the volume. On these slices, the volume data are reconstructed just as in standard

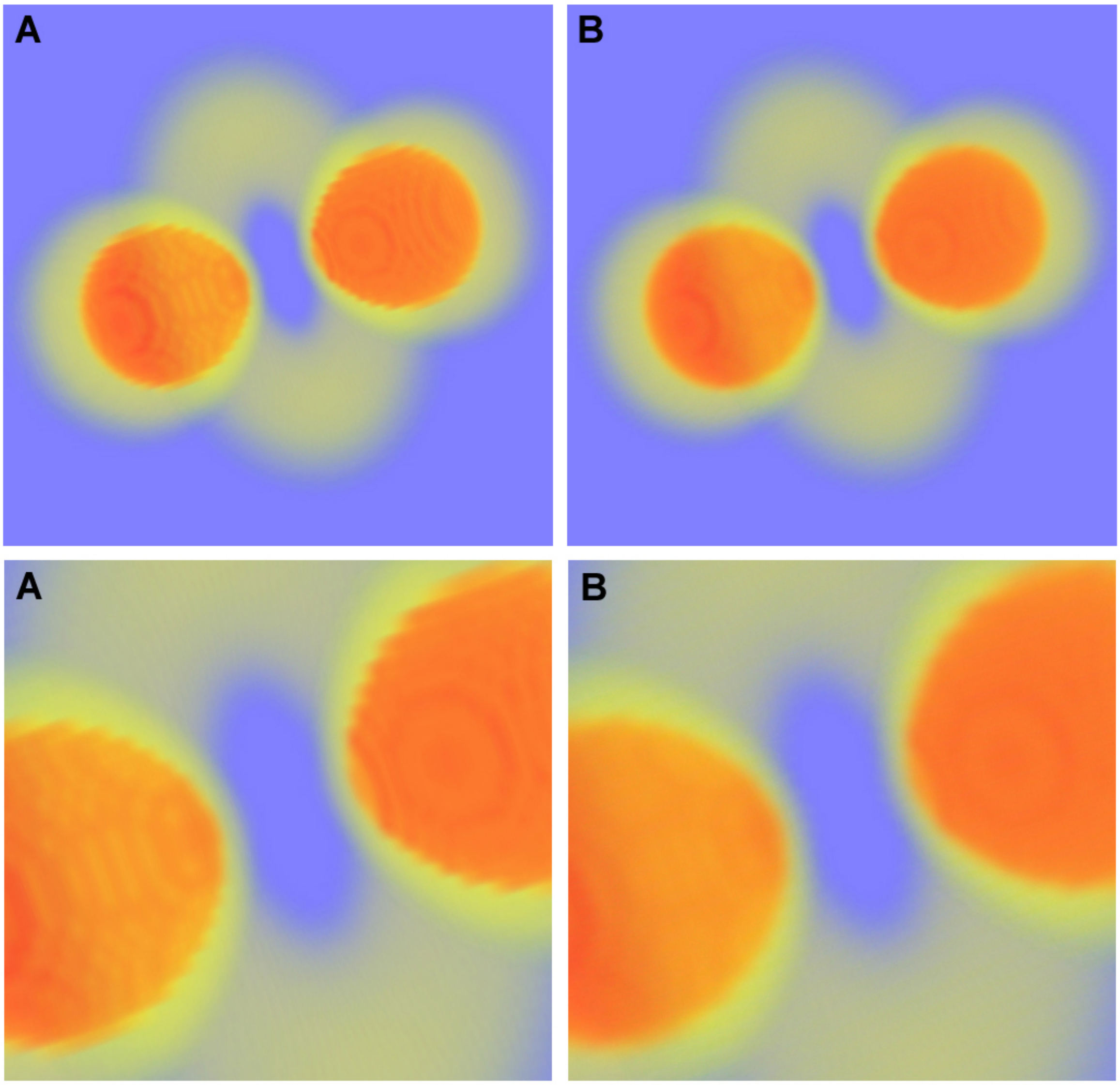


Figure 3.19: Using a high-quality reconstruction filter for volume rendering. This image compares bi-linear interpolation of object-aligned slices (A) with bi-cubic filtering using a B-spline filter kernel (B).

texture mapping, which allows to substitute the hardware-native interpolation with a higher-order reconstruction filter.

In the case of *object-aligned slices*, which use standard 2D textures, 2D convolution with a higher-order filter kernel can be used in order to improve reconstruction quality. For *view-aligned slices*, which require the volume data to be stored in a 3D texture map, 3D convolution with a higher-order filter can be used analogously.

Volume rendering

In order to generate volume renderings from individual texture-mapped slices, these slices have to be composited in either back-to-front or front-to-back order.

If high-quality filtering can be performed in a single rendering pass, depending on the filter kernel and graphics hardware features, substituting the hardware-native linear interpolation by the high-quality variant is trivial.

However, if the custom filter requires multiple rendering passes, each slice must be rendered into an off-screen buffer for reconstruction first. The best way to do this is to use the same viewing projection as for regular slice rendering, i.e., rendering the final view also during reconstruction. For most classification modes, this reconstruction step is still monochrome, e.g., reconstructing volume densities. After reconstruction, the resulting monochrome image is rendered into the output image with a 1-1 pixel mapping. During this mapping, classification

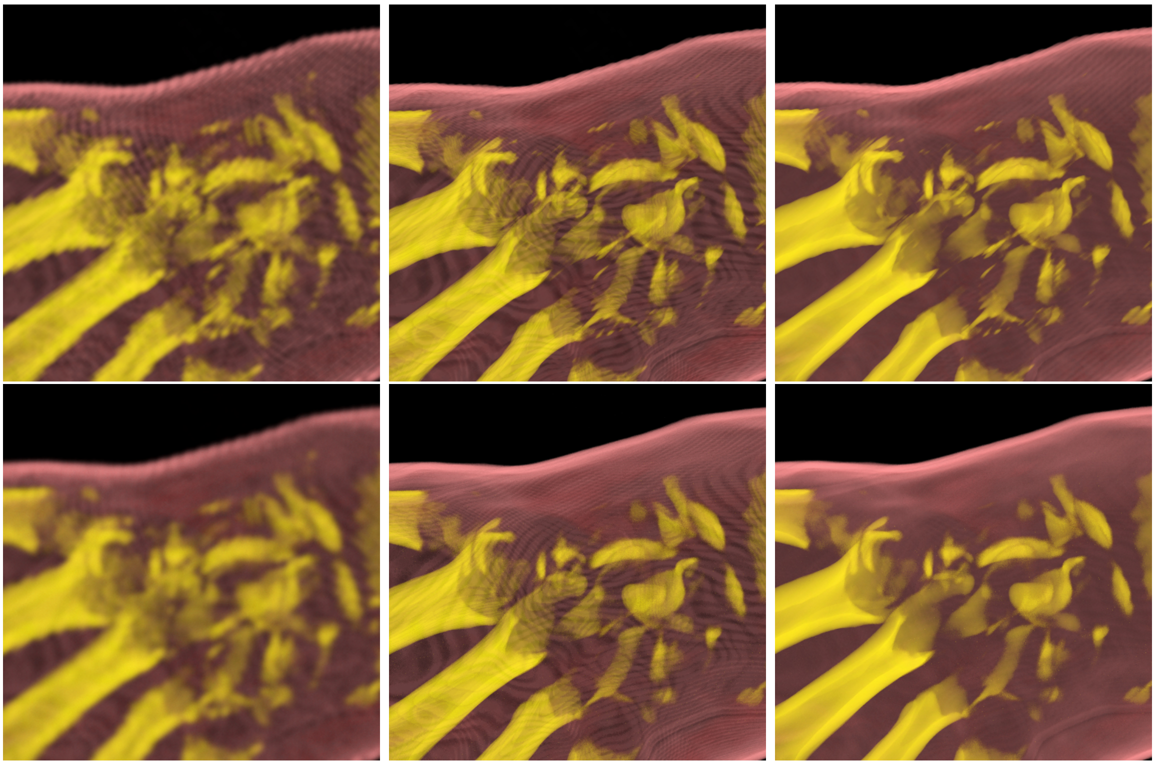


Figure 3.20: Hand (256x128x256) volume rendered with bi-linear (top row) vs. bi-cubic (bottom row) reconstruction of object-aligned slices. Left column with pre-classification; center column with post-classification; right column with pre-integration.

via the transfer function is performed on a per-pixel basis exactly as it would be done with hardware-native filtering.

Figure 3.19 shows an example of volume rendering with object-aligned slices filtered with a bi-cubic B-spline and classified with pre-classification.

Results

We now illustrate several results of cubic reconstruction for volume rendering and different classification modes.

Figures 3.20 and 3.21 compare bi-linear and bi-cubic slice interpolation for each of the major three classification modes: pre-classification, post-classification, and pre-integrated classification [24]. Pre-classification maps densities to colors and opacities before interpolation takes place, which leads to strong blurring artifacts (figure 3.20, first column), especially when the transfer function contains high frequencies. Post-classification interpolates densities and then maps the interpolation results to colors and opacities, which yields much sharper images (figure 3.20, center column). This preserves high frequencies in the transfer function much better. Pre-integrated classification decouples the frequencies in the transfer function entirely from the frequencies in the density volume, which yields even sharper images and is generally considered to be the best approach for low sampling rates (figure 3.20, right column).

The pre-integrated renderings also illustrate that even when pre-integration is used, cubic

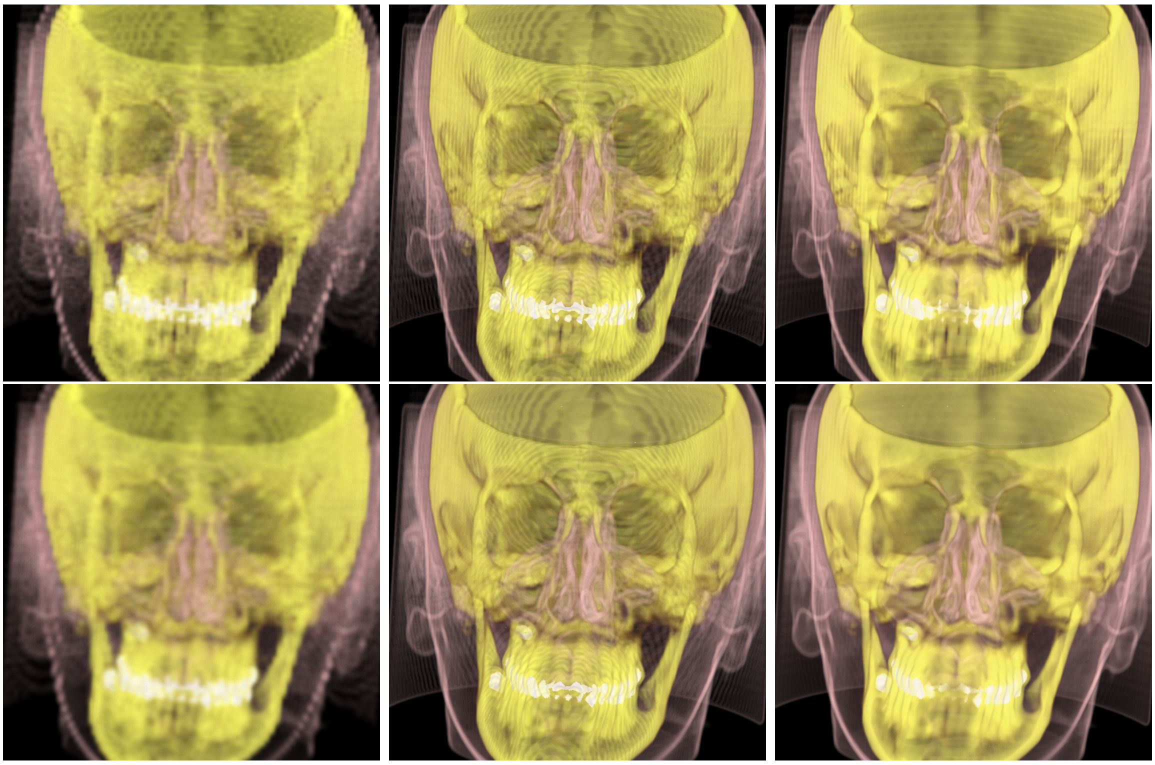


Figure 3.21: Head (128^3) volume rendered with bi-linear (top row) vs. bi-cubic (bottom row) reconstruction of object-aligned slices. Left column with pre-classification; center column with post-classification; right column with pre-integration.

filtering can still improve the final result significantly. See figures 3.22 (right column) and 3.23. The two approaches of pre-integration and higher-order slice filtering could be considered to improve quality in an orthogonal manner. Better filtering improves reconstruction quality within the individual slices, whereas pre-integration improves reconstruction quality with respect to the transfer function between slices, i.e., in the direction orthogonal to the slices themselves.

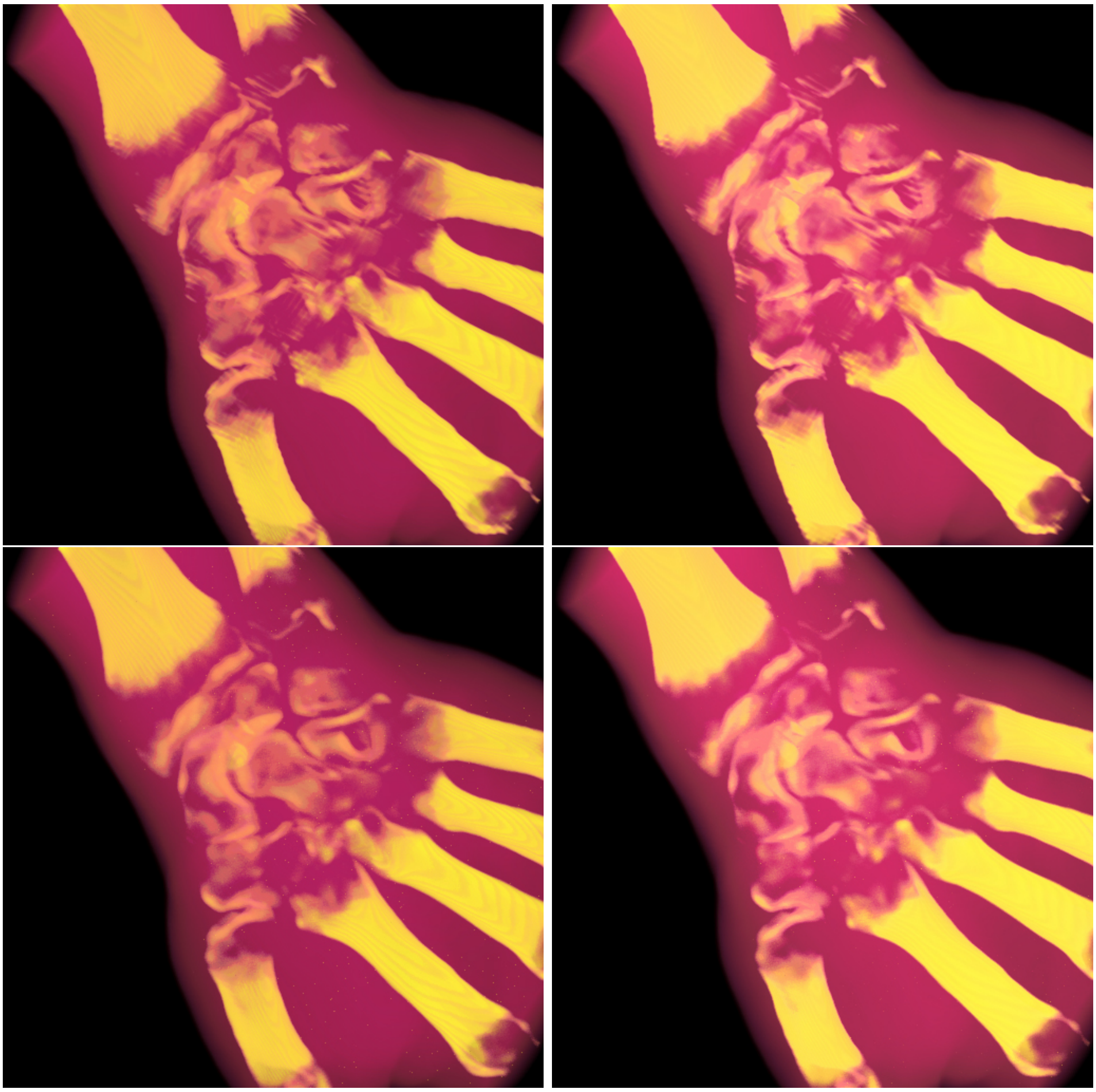


Figure 3.22: Hand (256x128x256) volume rendering with bi-linear (top row) vs. bi-cubic (bottom row) reconstruction of object-aligned slices. Left column with post-classification; right column with pre-integration.

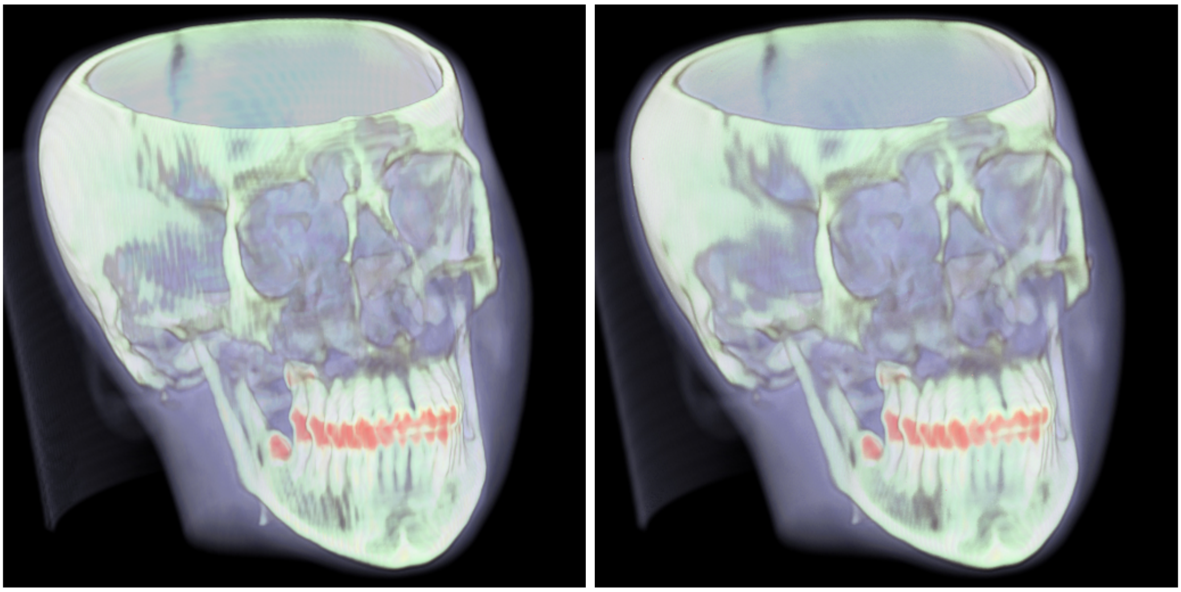


Figure 3.23: Head (128^3) volume rendering with pre-integration and bi-linear (left) and bi-cubic (right) reconstruction of object-aligned slices.

3.5 Error and Quality Considerations

This section is based on the paper *Quality Issues of Hardware-Accelerated High-Quality Filtering on PC Graphics Hardware* [35].

It summarizes several quality issues of the texture-based approach for convolution presented in previous sections of this chapter. Since this method uses multiple rendering passes, it is prone to precision and range problems related to the limited precision and range of intermediate computations and the color buffer, when floating point computations and color buffers are not employed. This is especially crucial on older consumer-level 3D graphics hardware, where usually only eight bits are stored per color component.

We estimate the accumulated error of several error sources, such as filter kernel quantization and discretization, precision of intermediate computations, and precision and range of intermediate results stored in the color buffer. We also describe two approaches for improving precision at the expense of a higher number of rendering passes. The first approach preserves higher internal precision over multiple passes that are forced to store intermediate results in the less-precise color buffer. The second approach employs hierarchical summation for attaining higher overall precision by using the available number of bits in a hierarchical fashion. Additionally, we consider issues such as the order of rendering passes that is crucial for avoiding potential range problems.

Precision and range problems of multi-pass rendering

Many recent real-time rendering algorithms are able to achieve high quality results by using many rendering passes. These methods accumulate intermediate results in the frame buffer in order to generate the final image.

One common problem shared by all of these approaches is the often limited range and precision of hardware frame buffers. Many graphics hardware architectures offer only eight bits per color component stored in the frame buffer, severely limiting the potential of color buffers for storing intermediate results of general computations. Additionally, the common $[0, 1]$ range even further complicates storing intermediate results in hardware color buffers, since not even a sign bit is available and large values cannot be stored directly. A common method for tackling the range problem is to employ scale and bias operations to fit the needed range into the constraints of the hardware. However, this further exacerbates the problems related to limited precision.

Recently, interest in higher precision and range for storing intermediate results of computations in graphics hardware has increased noticeably, especially since the introduction of real-time shading languages [124, 131]. Apart from simply extending the precision and range of frame buffers themselves, alternative approaches such as F-buffers [101] have also been suggested. Only with the most recent hardware architectures (ATI Radeon 9700 and NVIDIA GeForce FX), the process of moving towards floating-point color computations has begun.

Although the precision in color buffers of most available hardware is severely limited, recent hardware employs higher precision and range for internal computations before finally storing the unsigned eight-bit result in the frame buffer. The NVIDIA GeForce 3 and 4 officially use eight bits plus one sign bit internally, although our experimental results suggest that their internal precision is actually higher. Intermediate computations are performed in a range of $[-1, 1]$. The ATI Radeon 8500 internally uses twelve bits for the fractional part, plus four bits for integer part and sign, thus achieving an extended range of $[-8, 8]$.

Extended internal precision and range can be exploited for higher quality rendering by choosing an order of rendering passes that minimizes the impact of the limited external precision as much as possible. Additionally, higher internal precision can be preserved between passes by splitting up the intermediate results, only generating the final result in a final combination pass. The drawback of this, however, is a higher number of rendering passes.

We are especially interested in estimating the error incurred by limited precision and range in the approach for using arbitrary filter kernels for high-quality filtering on graphics hardware that has been presented in the preceding sections. We describe the sources of numerical error, and present results of a numerical simulation of the errors incurred by filter kernel quantization and discretization, according to different sampling resolutions and filter types. Kernel sampling parameters are a crucial issue in the approach we are interested in, since it samples and stores the actual continuous filter kernel into multiple texture maps. The algorithm treats the kernel as though it were continuous, although it is stored in a discrete representation, and reconstruction is used at run-time in order to retrieve weights from the "original" continuous filter function on-the-fly.

Estimating the error of hardware convolution

This section analyzes hardware-accelerated high-quality filtering with regard to the different sources of error (especially numerical error).

The filter convolution sum (equation 3.1) is evaluated in a multi-pass rendering algorithm, which is prone to artifacts due to precision and range limitations of the hardware color buffer and internal computations.

Error sources

Related to the evaluation of the filter convolution sum (equation 3.1), we distinguish the following sources of error.

First, error related to the filter kernel used and its representation in hardware:

- Since the filter kernel is sampled and stored in several texture maps, it has to be quantized to the bit-resolution of these textures. That is, the weights in the filter kernel are represented by b bits (where usually $b = 8$).
- Naturally, storing the filter kernel in texture maps also requires discretization (sampling), i.e., sampling the kernel at discrete locations. In the approach we are analyzing, the maximum texture resolution of the hardware can be used for each extent of unit size in the filter kernel. Thus, the width of the filter kernels that can be employed is not restricted by hardware texture size limits. However, in practice the sampling frequency used for the filter kernel itself is limited by the texture memory consumed.
- When retrieving weights from the filter kernel, the corresponding texture has to be queried, also employing reconstruction. For this, either point-sampling or the hardware-native linear interpolation is employed, which introduces further error in the filter weights that are actually used.
- The filter kernel also incurs an “inherent” non-numeric error, depending on its type and width. That is, a given kernel introduces a certain reconstruction error, even if we would be able to represent it as a continuous function, instead of a collection of discrete values. Möller et al. [109, 110, 111] present a framework for estimating filter kernel-native errors, as well as error bounds for several interesting types of filter kernels.

Second, error is introduced by the evaluation of the convolution sum in hardware using fixed-point arithmetic (usually mapping 1.0 to $2^b - 1$ instead of 2^b, and 0.0 to 0, in order to make maximum use of the available number of bits):

- Precision is lost in the multiplication of input samples with filter weights, since it introduces an error of its own (in contrast to addition, see below). The overall error therefore greatly depends on the precision with which multiplication is performed, and the number of multiplications needed for generating the final result, which, in our case, is equal to the number of rendering passes.
- Addition only propagates the error of its input operands if we assume that no clamping occurs due to range issues (see below). The error analysis in this section assumes that addition does not introduce new error by itself.
- If internal computations are carried out at a higher precision than the one available for storing intermediate results in the color buffer, further error that depends on the number of rendering passes required is introduced.

Third, additional – and usually quite severe – error may be introduced when the color buffer range is exceeded for intermediate results, leading to undesired clamping (i.e., values being forced into the $[0, 1]$ range by saturation). Therefore, it is crucial to choose an evaluation order of rendering passes that avoids leaving the available range for intermediate results. In practice, this means that intermediate results must never be below 0.0, or above 1.0. See section below for a description of how clamping artifacts can be avoided.

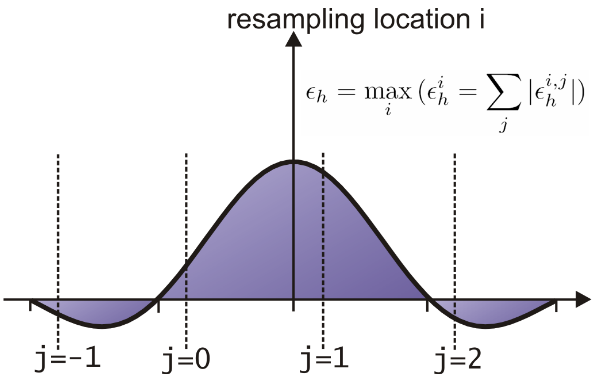

resampling location i
$\epsilon_h = \max_i (\epsilon_h^i = \sum_j |\epsilon_h^{i,j}|)$
j=-1
j=0
j=1
j=2

Figure 3.24: Estimating the error that is incurred by all weights retrieved from a texture map and contributing to the convolution sum.

Fixed-point representation

Computations on color values carried out by graphics hardware are usually done in a fixed-point representation. However, in order to use the entire dynamic range available for a given number of bits to represent the floating point range of $[0.0, 1.0]$ and still have an exact representation of 1.0, the interval $[0.0, 1.0]$ is usually [7, 120] mapped to $[0, 2^b - 1]$ instead of $[0, 2^b]$.

That is, as opposed to the usual fixed-point approach of mapping 1.0 to a power of two, it is mapped to the highest number representable with a certain number of bits. This maximizes utilization of the available number of bits, but somewhat complicates arithmetic operations in hardware, see below.

Mapping a floating-point color value x to a fixed-point color value $\bar{x}$ is thus done by:

$$\bar{x}_{(truncated)} = \lfloor x * (2^b - 1) \rfloor \tag{3.3}$$

if truncation is used, and by:

$$\bar{x}_{(rounded)} = \lfloor x * (2^b - 1) + 0.5 \rfloor \tag{3.4}$$

if rounding is used. Note that this special mapping leads to a slightly different quantization error than the one usually used, i.e., one lsb (least significant bit, i.e., 2^{-b}), see below.

Filter kernel error (ϵ_h)

We denote the error incurred by the representation of the filter kernel in texture maps by ϵ_h, and distinguish three sources of error that contribute to it.

First, due to the necessary quantization of input values to b bits, the following maximum error is introduced. If the high-precision input value is simply truncated to fit into the available number of bits, we get a quantization error $\epsilon_{q(truncated)} = \frac{1}{2^b - 1}$. Note that this is not exactly the usual one lsb, due to the special mapping described in the previous section. If rounding is employed, we get $\epsilon_{q(rounded)} = \frac{0.5}{2^b - 1}$. Further, error introduced by the limited sampling resolution (the discretization) and the reconstruction used for the filter kernel (either point-sampling via `GL_NEAREST`, or linear interpolation via `GL_LINEAR`) are two additional sources of numerical error that contribute to ϵ_h and must be considered.

Instead of actually considering these three sources of error separately, we calculate an estimation of the overall numerical error ϵ_h using a numerical simulation of the conditions corresponding to the hardware-accelerated algorithm. We sample the filter kernel at a specified resolution and quantize the sampled values to the number of bits that will be used in the actual texture maps. Using the reconstruction method that will be employed by the hardware to retrieve weights from the filter textures (point-sampling or linear interpolation), we derive approximately the same values that will be generated by the hardware and compare them with the corresponding reference values from the analytically represented filter.

In this way, we are able to estimate an error bound subsuming all three sources mentioned above for a single filter weight. In order to estimate the error introduced by the filter kernel into the entire evaluation of the filter convolution sum, however, we need to account for the error of all filter weights. For instance, filtering with a cubic kernel in one dimension uses four different weights, retrieved from the kernel at locations spaced one unit apart.

For a given resampling location i, the error incurred by all weights can be calculated as $\epsilon_h^i = \sum_j |\epsilon_h^{i,j}|$, with $\epsilon_h^{i,j}$ denoting the actual filter weights from the actual filter kernel under consideration that correspond to the given resampling location i, see figure 3.24. In order to estimate the error for all possible resampling locations, we calculate such ϵ_h^i at a high number of resampling locations and take the maximum, thus:

$$\epsilon_h = \max_i \,(\epsilon_h^i = \sum_j |\epsilon_h^{i,j}|) \tag{3.5}$$

Table 3.5 shows values of ϵ_h for certain scenarios, where ϵ_h subsumes the numerical errors due to quantization (ϵ_q), discretization, and filter kernel reconstruction in hardware. The corresponding filter functions are depicted in figure 3.3 in a previous section.

Computation error (ϵ_m)

In this section, we consider the error incurred by the evaluation of the convolution sum itself. This evaluation employs only two different kinds of arithmetic operations, namely addition and multiplication. We will see that the entire error due to the computation itself is introduced by the multiplication, ϵ_m.

Fixed-point arithmetic addition in hardware is usually simply done as $\bar{x} + \bar{y}$, since:

$$\bar{x} \oplus \bar{y} = x(2^b - 1) + y(2^b - 1) = (x + y)(2^b - 1) \tag{3.6}$$

The addition of two b bit values yields at most $b + 1$ bits in the result and there is no new error being introduced (assuming the result still fits into the available number of bits, thus avoiding undesired clamping).

Fixed-point arithmetic multiplication in hardware is usually done as $\lfloor (\bar{x}\bar{y})/(2^b - 1) + 0.5 \rfloor$, since:

$$\bar{x} \otimes \bar{y} = \frac{x(2^b - 1) * y(2^b - 1)}{2^b - 1} = (x * y)(2^b - 1) \tag{3.7}$$

The multiplication of two b bit values yields at most $2b$ bits in the result. Due to the division by $(2^b - 1)$ needed to normalize the result, a new error is introduced by the multiplication operation itself, even if the input values were exact.

The error that is introduced by the multiplication can be bounded by (for either truncated, or rounded results, respectively):

$$\epsilon_{m(truncated)} = \frac{2^b - 2}{(2^b - 1)^2} < \frac{1}{(2^b - 1)} \tag{3.8}$$

filter kernel type	tile resolution [1D samples/tile]	2D		3D	
		$255 * \epsilon_{h(box)}$	$255 * \epsilon_{h(lin)}$	$255 * \epsilon_{h(box)}$	$255 * \epsilon_{h(lin)}$
Cubic B-spline ($B = 1.0$, $C = 0.0$)	16	11.2351	5.5803	12.3400	7.2346
	32	6.1619	3.9219	7.1811	5.4758
	64	3.8838	3.3165	5.0478	7.0751
	128	2.8293	2.5841	4.4375	4.1687
	256	2.3145	2.2676	*n/a*	*n/a*
	512	2.0867	2.0867	*n/a*	*n/a*
Catmull-Rom ($B = 0.0$, $C = 0.5$)	16	25.3076	15.0835	32.6554	19.1470
	32	14.4100	10.8905	17.8089	14.3186
	64	8.7091	8.3325	11.2476	11.9767
	128	6.0308	7.1140	8.1039	10.5024
	256	4.9057	6.7169	*n/a*	*n/a*
	512	4.3535	6.3613	*n/a*	*n/a*
Blackman sinc (window width 4)	16	26.3035	15.4278	34.0124	20.0338
	32	14.3229	10.7960	17.9082	14.7748
	64	8.8159	8.3062	11.4056	12.0028
	128	6.0954	7.2064	8.1723	10.6158
	256	4.9426	6.6606	*n/a*	*n/a*
	512	4.3151	6.3229	*n/a*	*n/a*

Table 3.5: Filter kernel error bounds (ϵ_h) for different scenarios. All kernel weights have been quantized to eight bits, and the error bounds are absolute errors in a $[0, 1]$ domain, multiplied by 255. $\epsilon_{h(box)}$ uses a box filter for kernel reconstruction, $\epsilon_{h(lin)}$ linear interpolation. In the 2D case, the error has been estimated through 1024^2 resampling locations, whereas in the 3D case 256^3 resampling locations have been used (corresponding to the number of different values for i in equation 3.5). For comparison, just taking quantization to eight bits into account, a maximum absolute error of 0.5 would be incurred per filter weight, yielding error bounds of $8 = 0.5 * 16$ in two dimensions, and $32 = 0.5 * 64$ in three dimensions, respectively. Thus, the numbers in this table show that the error incurred by filter kernel representation in reality is much less than the overly conservative estimate of adding up the upper quantization error bounds of individual filter weights, e.g., 2.82 instead of 8 (bi-cubic B-spline, sampled with 128 samples per dimension and tile). In our experience, absolute errors of about 3–5 achieve results of sufficient optical quality. Usually, we are using 64 samples for a cubic B-spline, and 128 samples for Catmull-Rom splines and Blackman-windowed sincs, and simple nearest-neighbor interpolation for kernel reconstruction. *n/a* entries have not been measured, since 3D kernels of the corresponding sizes are infeasible, and the numerical simulation consumes a considerable amount of time.

$$\epsilon_{m(rounded)} = 0.5\frac{2^b - 2}{(2^b - 1)^2} < \frac{0.5}{(2^b - 1)} \tag{3.9}$$

For example, two cases interesting to us are:

$$\epsilon_{m(rounded,8)} = 0.0019608 \tag{3.10}$$

$$\epsilon_{m(rounded,12)} = 0.0001221 \tag{3.11}$$

for eight and twelve bits of precision, respectively.

Note that, although a division by a non-power-of-two value is theoretically necessary, the correct result – even including rounding – can for instance be generated without a division as follows (for $b = 8$ [7]):

```
i = x*y + 128;
r = ( i + ( i >> 8 ) ) >> 8;
```

Accumulated error

Now that we have bounded the individual errors involved, we can determine a worst-case bound for the overall evaluation of the filter convolution sum.

In the following, we assume that the input sample values are exact, the errors of the weights retrieved from the filter kernel are bounded by ϵ_h, and the error introduced by each multiplication in the convolution sum is bounded by ϵ_m.

The numerical errors introduced during evaluation of the filter convolution sum can be estimated as follows (denoting the number of rendering passes by N, and using $-\epsilon_m < \epsilon'_m < \epsilon_m$ and $-\epsilon_h < \epsilon'_h < \epsilon_h$ to denote actual errors corresponding to a given rendering pass as opposed to error bounds):

$$\sum_N f \otimes h_{texture} = \sum_N [f * (h + \epsilon'_h) + \epsilon'_m] \tag{3.12}$$

$$= \sum_N f * h + \sum_N f * \epsilon'_h + \sum_N \epsilon'_m \tag{3.13}$$

We now denote the overall error due to the numerical filter kernel representation, and the error introduced by the fixed-point multiplication by E_2 and E_3, respectively:

$$\sum_N f \otimes h_{texture} = \sum_N f * h + E_2 + E_3 \tag{3.14}$$

That is, these errors directly depend on the number of rendering passes needed for the evaluation of the convolution sum (N), and we get:

$$E_3 < N\epsilon_m \tag{3.15}$$

Assuming the function itself (f) is bounded by 1.0, we get:

$$E_2 < N\epsilon_h \tag{3.16}$$

Further introducing the error due to the filter itself (even if represented without numerical error, i.e., comparing the actual filter h to the ideal *sinc* filter) as E_1, we get:

$$\sum_N f \otimes h_{texture} = \sum_{-\infty}^{+\infty} f * sinc + E_1 + E_2 + E_3 \tag{3.17}$$

The error E_1 depends on the filter itself (i.e., its deviation from the behavior of a windowed sinc), and can be calculated using a Taylor series expansion of the convolution sum [110].

The most crucial restriction is the value of E_3, since it is entirely determined by the hardware. The other two errors can be chosen up to a certain extent. If possible, we try to do this in the following way:

- Choose the filter kernel such that $E_1 < E_3$, if possible.
- Choose the kernel texture resolution such that $E_2 < E_3$.

In practice, though, subjective visual judgment is the single most important criterion for selecting parameters, since eight to twelve bits of precision do not leave much space for considerations with conservative error bounds. In reality, visual results are still good where error estimation would suggest that too little precision has been used, and choosing the filter kernel sampling rate becomes most important.

Increasing precision for intermediate results

This section summarizes two approaches for gaining higher precision for the computation of intermediate results. The first approach strives to preserve higher internal precision supported by the graphics hardware across rendering passes that have to store intermediate results in the 8-bit frame buffer. It does not mandate any knowledge about the numerical subrange that will actually be used by a given set of rendering passes. The second approach assumes no higher internal precision, but performs summation in a hierarchical way that is also able to achieve higher precision results. However, it requires to know the numerical subrange that is actually used by a given set of rendering passes. Another approach that can be used to improve precision on lower-precision hardware has also been suggested recently [147].

Preserving internal precision across passes

Provided that internal computations are done by the hardware in higher precision than the external precision of the color buffer, a multi-pass approach can be used in order to preserve this precision across passes to a certain extent, even if temporaries need to be stored in the color buffer. In this context, we are exclusively dealing with addition. Multiplication is always performed with internal precision anyway, so we care about adding up the individual terms with higher precision than the frame buffer supports directly.

The basic idea is to perform the entire computation (i.e., evaluation of the convolution sum) twice. We denote the number of bits for internal computations by b and split it up into two bit-adjacent parts b_i and b_j, so that $b = b_i + b_j$. We also specify that the b_i bits contain the msb (most significant bit), and the b_j bits contain the lsb (least significant bit). If we denote the number of external bits (i.e., the precision of the frame buffer) by m, we choose $b_i = m$, and require that $b_j \leq m$. Since the resources of the frame buffer are more scarce than internal pixel paths, and calculating results in less precision than available for storing them makes no sense, $b \geq m$ of course also holds.

Now, we proceed by calculating two intermediate results and combining them afterwards. First, the desired computation is done for the part with b_i bits, storing the result in an off-screen buffer (either by rendering directly into a texture, or rendering into the framebuffer and copying it into a texture afterwards). Next, the same computation is done again, but this time for the part with b_j bits, yielding a certain number of carry bits, which we will be denoting by b_k. Finally, a single combination pass combines the two separate intermediate buffers, correctly taking the carry bits into account, and producing the final result. Thus, if the computation usually needs N rendering passes, we now need $2N + 1$ passes if we want to preserve the internal precision across passes. Note that if rendering directly to a texture is not supported, we only need to copy the frame buffer into a texture twice, independent of N. The reason for this is that we can do all computations in the frame buffer, except for the two input textures to the final combination pass, which need to be acquired.

In practice, we are most of all restricted by the number of carry bits that are generated and need to be stored. Over N passes, we create $b_k = \lfloor \log_2 N \rfloor$ carry bits, which have to fit into m bits together with the b_j base bits. Thus, $m \geq b_j + \lfloor \log_2 N \rfloor$ and if we want to preserve all internal bits, we are able to do so over at most $N = 2^{m-b_j+1} - 1$ passes.

We now give two examples with actual hardware-dependent numbers. First, on the GeForce 3 and 4, we assume that $b = 9$ and $m = 8$. Thus, $b_i = 8$, $b_j = 1$, and we can preserve all internal bits over at most $N = 255$ rendering passes, which would yield $b_k = 7$. However, we deem the associated performance impact for just one additional bit of precision too high. Second, on the Radeon 8500, we assume that $b = 12$ and $m = 8$. Thus, $b_i = 8$, $b_j = 4$, and we can preserve all internal bits over at most $N = 31$ rendering passes, which would yield $b_k = 4$. Further, for the interesting case of $N = 64$ (tri-cubic reconstruction of volume data), we can still preserve $m' = 10$ bits of internal precision ($m' \leq m$), which we still deem well worth the additional effort.

Another very implementation-dependent issue of the method outlined above, is how the internal results are actually split up into the two parts b_i and b_j, requiring bit shifting and masking, and how the combination pass is actually implemented, requiring bit shifting. We achieve bit shifting by possibly multiple multiplications with a scale factor, exploiting fixed scale and bias functionality of the graphics hardware. Factors less than one can also be achieved by multiplying with an arbitrary constant color, but multiplication with factors larger than one are not possible in this way, and require explicit support for scaling. The supported scale factors are hardware-dependent. Bit masking is usually not supported explicitly, and we achieve this by first shifting left, and then shifting right again, exploiting the automatic clamping to get rid of unwanted most significant bits. An alternative way would be to mask out the desired bits, and subtracting the undesired bits from the original value.

In the combination pass, the b_k carry bits produced in the computation corresponding to the b_j least significant bits have to be added to the part containing the result of the b_i most significant bits. For this, bit shifting is also required. That is, the carry bits have to be extracted from the intermediate result, and shifted into the proper position for addition.

Hierarchical summation

An approach for increasing the precision of intermediate computations, even given a limited internal precision, is to add results in a hierarchical manner.

Let b denote the number of bits available throughout the entire computation (i.e., both internally and externally). If we know that the filter weights used in a set of passes never exceed a certain threshold, we can pre-multiply all of these values in order to maximize usage of the available range, gaining “additional” bits of precision that would have been lost otherwise. The result of each of such a set of passes is accumulated in a corresponding off-screen rendering buffer. Renormalization to the actual range is then performed when compositing these intermediate results to generate the final image.

This approach is especially simple to realize in the case of 1:1 filtering (e.g., image processing), where there is only a very small number of filter weights. For example, the 16 passes associated with a 4x4 (2D) averaging filter with equal weights can be split up in batches of four passes, pre-multiplying the filter weights by 4 ("gaining" two bits of precision). There are four such batches, correspondingly generating four intermediate results. These have to be scaled by 0.25 and composited in a final combination pass.

Note that rendering to separate frame buffers for intermediate results can be achieved efficiently on many current graphics hardware architectures by using the `WGL_ARB_render_texture`, `WGL_ARB_pbuffer`, and `WGL_ARB_pixel_format` extensions, avoiding the slow `glCopyTexSubImage2D()` call.

Logarithm addition

We would like to briefly mention an idea brought up by Michael McCool [104] for gaining more range in multiplications. If we store the logarithm of input values to multiplications, we can add these values instead of multiplying them, and use a texture as lookup table for exponentiation in order to convert back to the actual output values when we need them.

Jim Blinn [8] also describes why floating point numbers are essentially a logarithmic representation.

Rendering pass order

The order of rendering passes is crucial to avoiding unintentional clamping of intermediate results against the $[0, 1]$ range of the frame buffer. We have therefore implemented a numerical simulator for the range behavior of a certain pass order. That is, we assume the worst-case input data of 1.0 everywhere, and accumulate the values contained in filter tiles for each sample location separately. The maximum and minimum values can be observed during accumulation. Using this facility makes it possible to ascertain beforehand whether a certain pass order is able to avoid clamping errors or not. If the maximum and minimum values never go above 1.0 or 0.0, respectively, during accumulation, the pass order can be used at run-time.

Additionally, if it is not possible to find a pass order that avoids clamping, we split up some filter tiles into two sub-tiles, one of them containing the larger values of the tile, the other one containing the smaller values. These two separate tiles can be inserted into the pass order at arbitrary (and non-adjacent) locations, which allows to avoid clamping, even if this would not have been possible otherwise. Naturally, this increases the number of rendering passes that are required.

Rendering pass bias

When we want to directly reconstruct gradients, instead of original function values, bias values have to be added in each pass. For example, the vector component values in normalized gradients are between -1 and 1. However, in hardware rendering, this is mapped to a $[0.0, 1.0]$ range by scaling by 0.5, and adding a bias of 0.5.

We add individual bias values in each rendering pass, which altogether sum up to the required overall bias of 0.5. The reason for this is once again to avoid clamping errors between passes. A single addition of a single 0.5 bias allows no fine-control over when (during passes) negative values can be avoided by adding a small bias, but simultaneously not adding so much as to go over 1.0.

Section summary and conclusions

In this section, we have described the different sources of numerical error that are relevant in hardware-accelerated high-quality filtering with arbitrary filter kernels sampled into multiple texture maps.

The numerical simulations that we have presented help to determine an order of rendering passes that avoids unintentional clamping of intermediate results, and allow to estimate the interdependence of filter kernel sampling resolution, filter kernel type, and quantization. As was to be expected from the frequencies contained in those filter kernels, the cubic B-spline causes the least numerical problems, followed by the Catmull-Rom spline and a Blackman-windowed sinc, respectively.

We have also estimated the overall error incurred during evaluation of the filter convolution sum in hardware, taking the above-mentioned error sources into account.

Chapter 4

Volume Rendering of Segmented Data

This chapter is based on the paper *High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware* [34].

One of the most important goals in volume rendering, especially when dealing with medical data, is to be able to visually separate and selectively enable specific objects of interest contained in a single volumetric data set. A very powerful approach to facilitate the perception of individual objects is to create explicit object membership information via *segmentation* [159]. The process of segmentation determines a set of voxels that belong to a given object of interest, usually in the form of one or several *segmentation masks*. There are two major ways of representing segmentation information in masks. First, each object can be represented by a single binary segmentation mask, which determines for each voxel whether it belongs to the given object or not. Second, an object ID volume can specify segmentation information for all objects in a single volume, where each voxel contains the ID of the object it belongs to. These masks can then be used to selectively render only some of the objects contained in a single data set, or render different objects with different optical properties such as transfer functions, for example.

Other approaches for achieving visual distinction of objects are for example rendering multiple semi-transparent isosurfaces, or direct volume rendering with an appropriate transfer function. In the latter approach, multi-dimensional transfer functions [66, 69] have proven to be especially powerful in facilitating the perception of different objects. However, it is often the case that a single rendering method or transfer function does not suffice in order to distinguish multiple objects of interest according to a user's specific needs, especially when spatial information needs to be taken into account. Non-photorealistic volume rendering methods [16, 22, 99] have also proven to be promising approaches for achieving better perception of individual objects.

An especially powerful approach is to combine different non-photorealistic and traditional volume rendering methods in a single volume rendering. When segmentation information is available, different objects can be rendered with individual per-object rendering modes, which allows to use specific modes for structures they are well suited for, as well as separating *focus* from *context*. Even further, different objects can be rendered with their own individual compositing mode, combining the contributions of all objects with a single global compositing mode. This two-level approach to object compositing can facilitate object perception very effectively and is known as *two-level volume rendering* [43, 44].

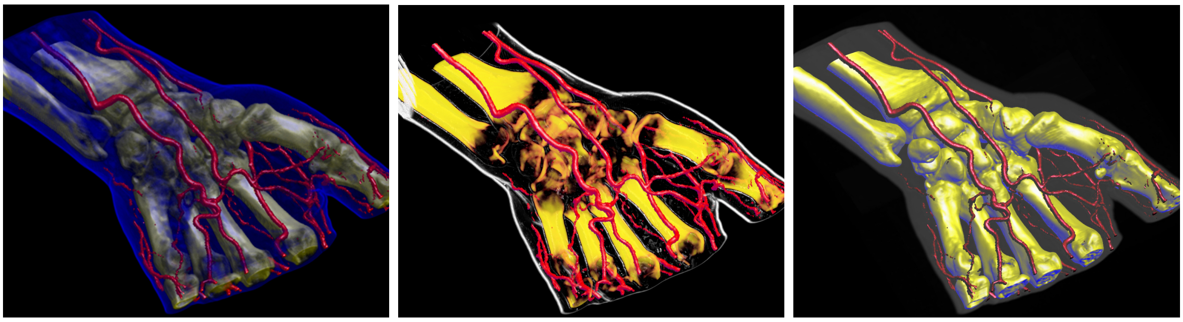


Figure 4.1: Segmented hand data set (256x128x256) with three objects: skin, blood vessels, and bone. Two-level volume rendering integrates different transfer functions, rendering and compositing modes: (left) all objects rendered with shaded DVR; the skin partially obscures the bone; (center) skin rendered with non-photorealistic contour rendering and MIP compositing, bones rendered with DVR, vessels with tone shading; (right) skin rendered with MIP, bones with tone shading, and vessels with shaded isosurfacing; the skin merely provides context.

Contribution

Integrating segmentation information and multiple rendering modes with different sets of parameters into a fast high-quality volume renderer is not a trivial problem, especially in the case of consumer hardware volume rendering, which tends to only be fast when all or most voxels can be treated identically. On such hardware, one would also like to use a single segmentation mask volume in order to use a minimal amount of texture memory. Graphics hardware cannot easily interpolate between voxels belonging to different objects, however, and using the segmentation mask without filtering gives rise to artifacts. Thus, one of the major obstacles in such a scenario is filtering object boundaries in order to attain high quality in conjunction with consistent fragment assignment and without introducing non-existent object IDs due to interpolation.

In this chapter, we show how segmented volumetric data sets can be rendered efficiently and with high quality on current consumer graphics hardware. The segmentation information for object distinction can be used at multiple levels of sophistication, and we describe how all of these different possibilities can be integrated into a single coherent hardware volume rendering framework.

First, different objects can be rendered with the same rendering technique (e.g., DVR), but with different transfer functions. Separate per-object transfer functions can be applied in a single rendering pass even when object boundaries are filtered during rendering. On an ATI Radeon 9700, up to eight transfer functions can be folded into a single rendering pass with linear boundary filtering. If boundaries are only point-sampled, e.g., during interaction, an arbitrary number of transfer functions can be used in a single pass. However, the number of transfer functions with boundary filtering in a single pass is no conceptual limitation and increases trivially on architectures that allow more instructions in the fragment shader.

Second, different objects can be rendered using different hardware fragment shaders. This allows easy integration of methods as diverse as non-photorealistic and direct volume rendering, for instance. Although each distinct fragment shader requires a separate rendering pass, multiple objects using the same fragment shader with different rendering parameters can effectively be combined into a single pass. When multiple passes cannot be avoided, the cost of individual passes is reduced drastically by executing expensive fragment shaders only for those fragments active in a given pass. These two properties allow highly interactive rendering of segmented data sets, since even for data sets with many objects usually only a couple of different rendering modes are employed. We have implemented direct volume rendering with post-classification, pre-integrated classification [24], different shading modes, non-polygonal isosurfaces, and maximum intensity projection. See figures 4.1 and 4.2 for example images. In addition to non-photorealistic contour enhancement [16] (figure 4.1, center; figure 4.2, skull), we have also used a volumetric adaptation of tone shading [28] (figure 4.1, right), which improves depth perception in contrast to standard shading.

Finally, different objects can also be rendered with different compositing modes, e.g., alpha blending and maximum intensity projection (MIP), for their contribution to a given pixel. These per-object compositing modes are object-local and can be specified independently for each object. The individual contributions of different objects to a single pixel can be combined via a separate global compositing mode. This two-level approach to object compositing [43, 44] has proven to be very useful in order to improve perception of individual objects.

In summary, the major novel contributions of this chapter are:

- A systematic approach to minimizing both the number of rendering passes and the performance cost of individual passes when rendering segmented volume data with high quality on current GPUs. Both filtering of object boundaries and the use of different rendering parameters such as transfer functions do not prevent using a single rendering pass for multiple objects. Even so, each pass avoids execution of the corresponding potentially expensive fragment shader for irrelevant fragments by exploiting the early z-test. This reduces the performance impact of the number of rendering passes drastically.

- An efficient method for mapping a single object ID volume to and from a domain where filtering produces correct results even when three or more objects are present in the volume. The method is based on simple 1D texture lookups and able to map and filter blocks of four objects simultaneously.

- An efficient object-order algorithm based on simple depth and stencil buffer operations that achieves correct compositing of objects with different per-object compositing modes and an additional global compositing mode. The result is conceptually identical to being able to switch compositing modes for any given group of samples along the ray for any given pixel.

4.1 Segmented Data Representation

For rendering purposes, we simply assume that in addition to the usual data such as a density and an optional gradient volume, a *segmentation mask volume* is also available. If embedded objects are represented as separate masks, we combine all of these masks into a single volume that contains a single object ID for each voxel in a pre-process. Hence we will also be calling this segmentation mask volume the *object ID volume*. IDs are simply enumerated consecutively starting with one, i.e., we do not assign individual bits to specific objects. ID zero is reserved (see later sections).

The object ID volume consumes one byte per voxel and is either stored in its own 3D texture in the case of view-aligned slicing, or in additional 2D slice textures for all three slice stacks in the case of object-aligned slicing. With respect to resolution, we have used the same resolution as the original volume data, but all of the approaches we describe could easily be used for volume and segmentation data of different resolutions.

4.2 Rendering Segmented Data

In order to render a segmented data set, we determine object membership of individual fragments by filtering object boundaries in the hardware fragment shader (section 4.3). Object membership determines which transfer function, rendering, and compositing modes should be used for a given fragment. See figure 4.3 for an example of three segmented objects rendered with per-object rendering modes and transfer functions.

We render the volume in a number of rendering passes that is basically independent of the number of contained objects. It most of all depends on the required number of different hardware configurations that cannot be changed during a single pass, i.e., the fragment shader and compositing mode. Objects that can share a given configuration can be rendered in a

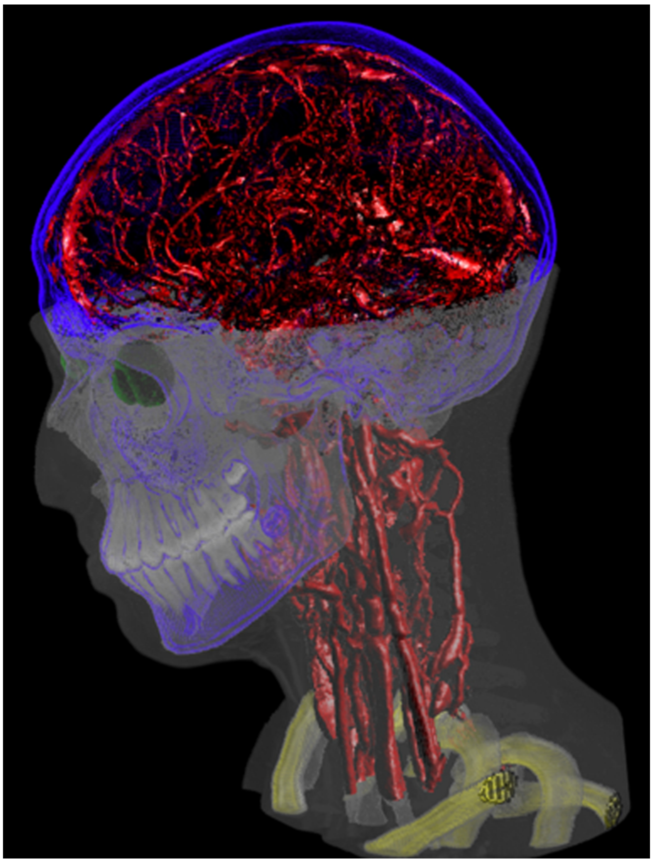


Figure 4.2: Segmented head and neck data set (256x256x333) with six different enabled objects. The skin and teeth are rendered as MIP with different intensity ramps, the blood vessels and eyes are rendered as shaded DVR, the skull uses contour rendering, and the vertebrae use a gradient magnitude-weighted transfer function with shaded DVR. A clipping plane has been applied to the skin object.

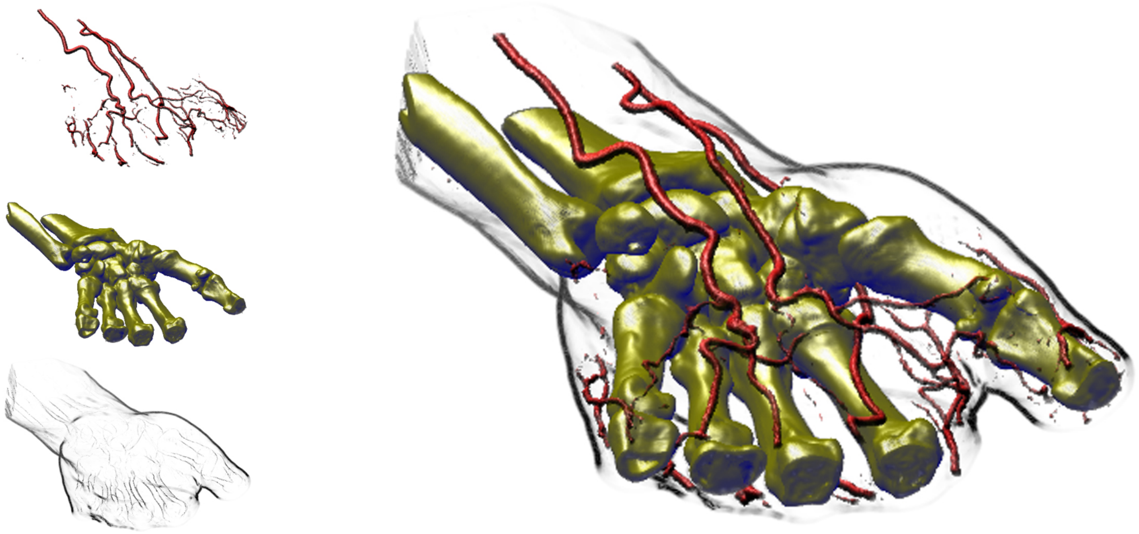


Figure 4.3: CT scan of a human hand (256x128x256) with three segmented objects (skin, blood vessels, and bone structure). The skin is rendered with contour enhancement, the vessels with shaded DVR, and the bones with tone shading.

single pass. This also extends to the application of multiple per-object transfer functions (section 4.3) and thus the actual number of rendering passes is usually much lower than the number of objects or transfer functions. It depends on several major factors:

Enabled objects. If all the objects rendered in a given pass have been disabled by the user, the entire rendering pass can be skipped. If only some of the objects are disabled, the number of passes stays the same, independent of the order of object IDs. Objects are disabled by changing a single entry of a 1D lookup texture. Additionally, per-object clipping planes can be enabled. In this case, all objects rendered in the same pass are clipped identically, however.

Rendering modes. The rendering mode, implemented as an actual hardware fragment shader, determines what and how volume data is re-sampled and shaded. Since it cannot be changed during a single rendering pass, another pass must be used if a different fragment shader is required. However, many objects often use the same basic rendering mode and thus fragment shader, e.g., DVR and isosurfacing are usually used for a large number of objects.

Transfer functions. Much more often than the basic rendering mode, a change of the transfer function is required. For instance, all objects rendered with DVR usually have their own individual transfer functions. In order to avoid an excessive number of rendering passes due to simple transfer function changes, we apply multiple transfer functions to different objects in a single rendering pass while still retaining adequate filtering quality (section 4.3).

Compositing modes. Although usually considered a part of the rendering mode, compositing is a totally separate operation in graphics hardware. Where the basic rendering mode is determined by the fragment shader, the compositing mode is specified as blend function and equation in OpenGL, for instance. It determines how already shaded fragments are combined with pixels stored in the frame buffer. Changing the compositing mode happens even more infrequently than changing the basic rendering mode, e.g., alpha blending is used in conjunction with both DVR and tone shading.

Different compositing modes per object also imply that the (conceptual) ray corresponding to a single pixel must be able to combine the contribution of these different modes (figure 4.10). Especially in the context of texture-based hardware volume rendering, where no actual rays exist and we want to obtain the same result with an object-order approach instead, we have to use special care when compositing. The contributions of individual objects to a given pixel should not interfere with each other, and are combined with a single global compositing mode.

In order to ensure correct compositing, we are using two render buffers and track the current compositing mode for each pixel. Whenever the compositing mode changes for a given pixel, the already composited part is transferred from the *local compositing buffer* into the *global compositing buffer*. Section 4.4 shows that this can actually be done very efficiently without explicitly considering individual pixels, while still achieving the same compositing behavior as a ray-oriented image-order approach, which is crucial for achieving high quality. For faster rendering we allow falling back to single-buffer compositing during interaction (figure 4.11).

The basic rendering loop

We will now outline the basic rendering loop that we are using for each frame. Table 4.1 gives a high-level overview.

Although the user is dealing with individual objects, we automatically collect all objects that can be processed in the same rendering pass into an *object set* at the beginning of each frame. For each object set, we generate an *object set membership texture*, which is a 1D lookup table that determines the objects belonging to the set. In order to further distinguish different transfer functions in a single object set, we also generate 1D *transfer function assignment textures*. Both of these types of textures are shown in figure 4.5 and described in sections 4.2 and 4.3.

After this setup, the entire slice stack is rendered. Each slice must be rendered for every object set containing an object that intersects the slice, which is determined in a pre-process.

```
DetermineObjectSets();
CreateObjectSetMembershipTextures();
CreateTFAssignmentTextures();
FOR each slice DO
  TransferLocalBufferIntoGlobalBuffer();
  ClearTransferredPixelsInLocalBuffer();
  RenderObjectIdDepthImageForEarlyZTest();
  FOR each object set with an object in slice DO
    SetupObjectSetFragmentRejection();
    SetupObjectSetTFAssignment();
    ActivateObjectSetFragmentShader();
    ActivateObjectSetCompositingMode();
    RenderSliceIntoLocalBuffer();
```

Table 4.1: The basic rendering loop that we are using. Object set membership can change every time an object's rendering or compositing mode is changed, or an object is enabled or disabled.

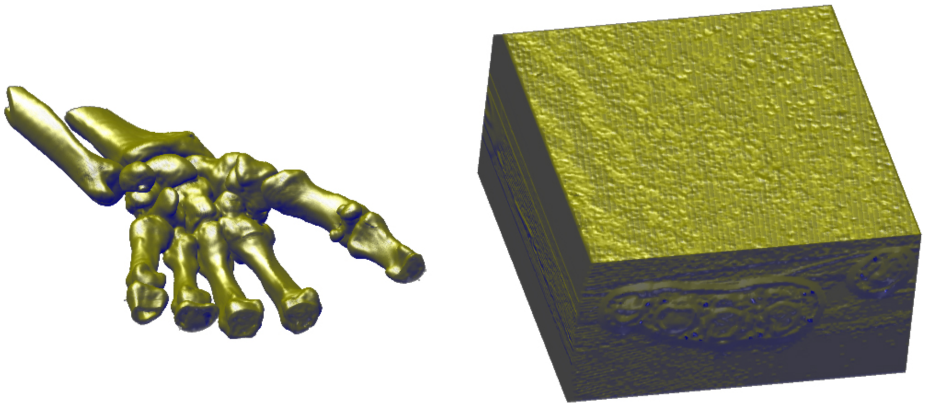


Figure 4.4: In order to render the bone structure shown on the left, many voxels need to be culled. The early z-test allows to avoid evaluating shading equations for culled voxels. If it is not employed, performance will correspond to shading all voxels as shown on the right.

In the case of 3D volume textures, all slices are always assumed to be intersected by all objects, since they are allowed to cut through the volume at arbitrary angles. If there is more than a single object set for the current slice, we optionally render all object set IDs of the slice into the depth buffer before rendering any actual slice data. This enables us to exploit the early z-test during all subsequent passes for each object set, see below. For performance reasons, we never use object ID filtering in this pass, which allows only conservative fragment culling via the depth test. Exact fragment rejection is done in the fragment shader.

We proceed by rendering actual slice data. Before a slice can be rendered for an object set, the fragment shader and compositing mode corresponding to this set must be activated. Using the two types of textures mentioned above, the fragment shader filters boundaries, rejects fragments not corresponding to the current pass, and applies the correct transfer function.

In order to attain two compositing levels, slices are rendered into a local buffer, as already outlined above. Before rendering the current slice, those pixels where the local compositing mode differs from the previous slice are transferred from the local into the global buffer using the global compositing mode. After this transfer, the transferred pixels are cleared in the local buffer to ensure correct local compositing for subsequent pixels. In the case when only a single compositing buffer is used for approximate compositing, the local to global buffer transfer and clear are not executed.

Finally, if the global compositing buffer is separate from the viewing window, it has to be transferred once after the entire volume has been rendered.

Conservative fragment culling via early z-test

On current graphics hardware, it is possible to avoid execution of the fragment shader for fragments where the depth test fails as long as the shader does not modify the depth value of the fragment. This early z-test is crucial to improving performance when multiple rendering passes have to be performed for each slice.

If the current slice's object set IDs have been written into the depth buffer before, see above, we conservatively reject fragments not belonging to the current object set even before the corresponding fragment shader is started. In order to do this, we use a depth test of `GL_EQUAL` and configure the vertex shader to generate a constant depth value for each fragment that exactly matches the current object set ID. Figure 4.4 graphically illustrates the performance difference of using the early z-test as opposed to also shading voxels that will be culled.

Excluding individual fragments from processing by an expensive fragment shader via the early z-test is also crucial in the context of GPU-based ray casting in order to be able to terminate rays individually [78].

Fragment shader operations

Most of the work in volume renderers for consumer graphics hardware is done in the fragment shader, i.e., at the granularity of individual fragments and, ultimately, pixels. In contrast to approaches using lookup tables, i.e., paletted textures, we are performing all shading operations procedurally in the fragment shader. However, we are most of all interested in the operations that are required for rendering segmented data. The two basic operations in the fragment shader with respect to the segmentation mask are fragment rejection and per-fragment application of transfer functions:

Fragment rejection. Fragments corresponding to object IDs that cannot be rendered in the current rendering pass, e.g., because they need a different fragment shader or compositing mode, have to be rejected. They, in turn, will be rendered in another pass, which uses an appropriately adjusted rejection comparison.

For fragment rejection, we do not compare object IDs individually, but use 1D lookup textures that contain a binary membership status for each object (figure 4.5, left). All objects that can be rendered in the same pass belong to the same object set, and the corresponding object set membership texture contains ones at exactly those texture coordinates corresponding to the IDs of these objects, and zeros everywhere else. The re-generation of these textures at the beginning of each frame, which is negligible in terms of performance, also makes turning individual objects on and off trivial. Exactly one object set membership texture is active for a given rendering pass and makes the task of fragment rejection trivial if the object ID

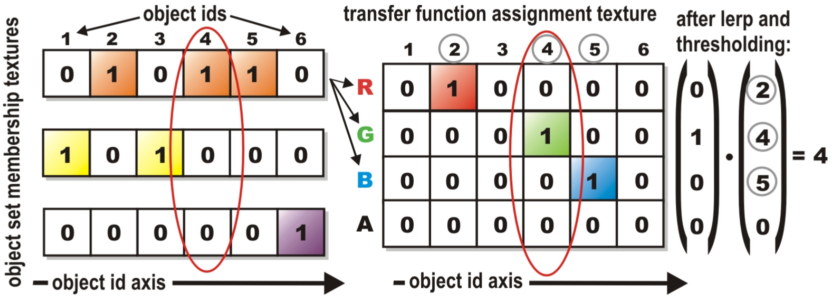


Figure 4.5: Object set membership textures (left; three 1D intensity textures for three sets containing three, two, and one object, respectively) contain a binary membership status for each object in a set that can be used for filtering object IDs and culling fragments. Transfer function assignment textures (right; one 1D RGBA texture for distinction of four transfer functions) are used to filter four object boundaries simultaneously and determine the corresponding transfer function via a simple dot product.

volume is point-sampled.

When object IDs are filtered, it is also crucial to map individual IDs to zero or one before actually filtering them. Details are given in section 4.3, but basically we are using object set membership textures to do a binary classification of input IDs to the filter, and interpolate after this mapping. The result can then be mapped back to zero or one for fragment rejection.

Per-fragment transfer function application. Since we apply different transfer functions to multiple objects in a single rendering pass, the transfer function must be applied to individual fragments based on their density value and corresponding object ID. Instead of sampling multiple one-dimensional transfer function textures, we sample a single global two-dimensional transfer function texture (figure 4.6). This texture is not only shared between all objects of an object set, but also between all object sets. It is indexed with one texture coordinate corresponding to the object ID, the other one to the actual density.

Because we would like to filter linearly along the axis of the actual transfer function, but use point-sampling along the axis of object IDs, we store each transfer function twice at adjacent locations in order to guarantee point-sampling for IDs, while we are using linear interpolation for the entire texture. We have applied this scheme only to 1D transfer functions, but general 2D transfer functions could also be implemented via 3D textures of just a few layers in depth, i.e., the number of different transfer functions.

We are using an extended version of the pixel-resolution filter that we employ for fragment rejection in order to determine which of multiple transfer functions in the same rendering pass a fragment should actually use. Basically, the fragment shader uses multiple RGBA transfer function assignment textures (figure 4.5, right) for both determining the transfer function and rejecting fragments, instead of a single object set membership texture with only a single color channel. Each one of these textures allows filtering the object ID volume with respect to four object boundaries simultaneously. A single lookup yields binary membership classification of a fragment with respect to four objects. The resulting RGBA membership vectors can then be interpolated directly. The main operation for mapping back the result to an object ID is a simple dot product with a constant vector of object IDs. If the result is the non-existent object ID of zero, the fragment needs to be rejected. The details are described in section 4.3.

This concept can be extended trivially to objects sharing transfer functions by using transfer function IDs instead of object IDs. The following two sections will now describe filtering of object boundaries at sub-voxel precision in more detail.

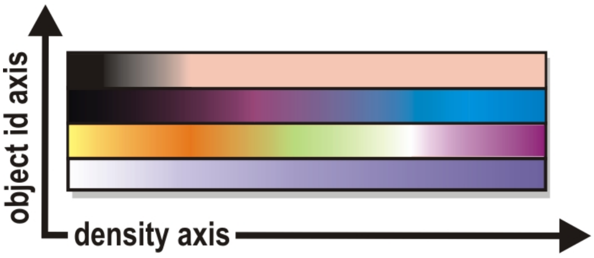


Figure 4.6: Instead of multiple one-dimensional transfer functions for different objects, we are using a single global two-dimensional transfer function texture. After determining the object ID for the current fragment via filtering, the fragment shader appropriately samples this texture with $(density, object\_id)$ texture coordinates.

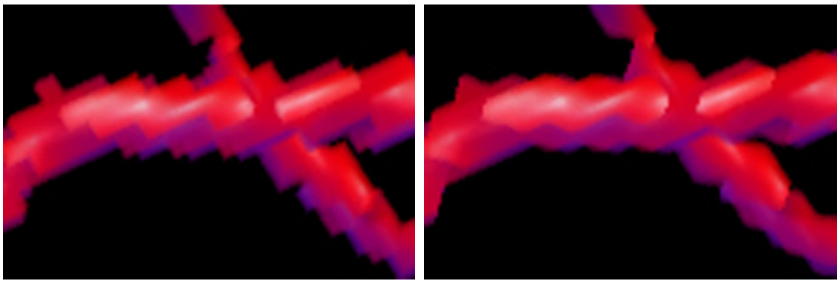


Figure 4.7: Object boundaries with voxel resolution (left) vs. object boundaries determined per-fragment with linear filtering (right).

4.3 Boundary Filtering

One of the most crucial parts of rendering segmented volumes with high quality is that the object boundaries must be calculated during rendering at the pixel resolution of the output image, instead of the voxel resolution of the segmentation volume. Figure 4.7 (left) shows that simply point-sampling the object ID texture leads to object boundaries that are easily discernible as individual voxels. That is, simply retrieving the object ID for a given fragment from the segmentation volume is trivial, but causes artifacts. Instead, the object ID must be determined via filtering for each fragment individually, thus achieving pixel-resolution boundaries.

Unfortunately, filtering of object boundaries cannot be done directly using the hardware-native linear interpolation, since direct interpolation of numerical object IDs leads to incorrectly interpolated intermediate values when more than two different objects are present. When filtering object IDs, a threshold value s_t must be chosen that determines which object a given fragment belongs to, which is essentially an iso-surfacing problem.

However, this cannot be done if three or more objects are contained in the volume, which is illustrated in the top row of figure 4.8. In that case, it is not possible to choose a single s_t for the entire volume. The crucial observation to make in order to solve this problem is that the segmentation volume must be filtered as a successive series of binary volumes in order to achieve proper filtering [154], which is shown in the second row of figure 4.8. Mapping all object IDs of the current object set to 1.0 and all other IDs to 0.0 allows using a global threshold value s_t of 0.5. We of course do not want to store these binary volumes explicitly, but perform this mapping on-the-fly in the fragment shader by indexing the *object set membership texture* that is active in the current rendering pass. Filtering in the other passes simply uses an alternate binary mapping, i.e., other object set membership textures.

One problem with respect to a hardware implementation of this approach is that texture filtering happens before the sampled values can be altered in the fragment shader. Therefore, we perform filtering of object IDs directly in the fragment shader. Note that our approach could in part also be implemented using texture palettes and hardware-native linear interpolation, with the restriction that not more than four transfer functions can be applied in a single rendering pass. However, we have chosen to perform all filtering in the fragment shader in order to create a coherent framework with a potentially unlimited number of transfer functions in a single rendering pass and prepare for the possible use of cubic boundary filtering in the future.

After filtering yields values in the range $[0.0, 1.0]$, we once again come to a binary decision whether a given fragment belongs to the current object set by comparing with a threshold value of 0.5 and rejecting fragments with an interpolated value below this threshold (figure 4.8, third row).

Actual rejection of fragments is done using the `KIL` instruction of the hardware fragment shader that is available in the `ARB_fragment_program` OpenGL extension, for instance. It can also be done by mapping the fragment to RGBA values constituting the identity with respect to the current compositing mode (e.g., an alpha of zero for alpha blending), in order to not alter the frame buffer pixel corresponding to this fragment.

Linear boundary filtering. For object-aligned volume slices, bi-linear interpolation is done by setting the hardware filtering mode for the object ID texture to nearest-neighbor and sampling it four times with offsets of whole texels in order to get access to the four ID values needed for interpolation. Before actual interpolation takes place, the four object IDs are individually mapped to 0.0 or 1.0, respectively, using the current object set membership texture.

We perform the actual interpolation using a variant of texture-based filtering [36], which proved to be both faster and use fewer instructions than using `LRP` instructions. With this approach, bi-linear weight calculation and interpolation can be reduced to just one texture fetch and one dot product. When intermediate slices are interpolated on-the-fly [133], or view-aligned slices are used, eight instead of four input IDs have to be used in order to perform tri-linear interpolation.

Combination with pre-integration. The combination of pre-integration [24] and high-quality clipping has been described recently [137]. Since our filtering method effectively reduces the segmentation problem to a clipping problem on-the-fly, we are using the same approach after we have mapped object IDs to 0.0 or 1.0, respectively. In this case, the interpolated binary values must be used for adjusting the pre-integration lookup.

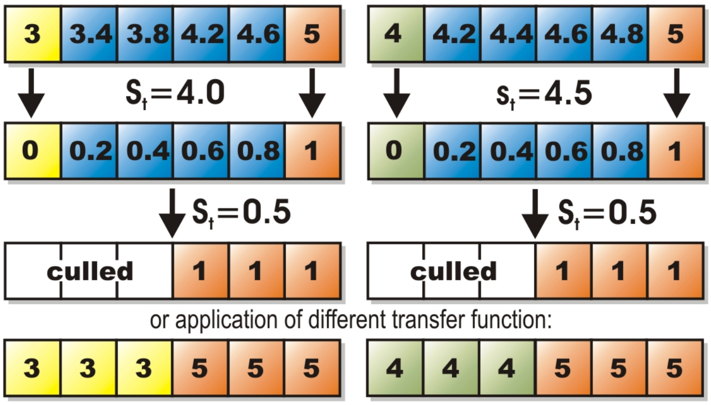


Figure 4.8: Each fragment must be assigned an exactly defined object ID after filtering. Here, IDs 3, 4, and 5 are interpolated, yielding the values shown in blue. Top row: choosing a single threshold value s_t that works everywhere is not possible for three or more objects. Second row: object IDs must be converted to 0.0 or 1.0 in the fragment shader before interpolation, which allows using a global s_t of 0.5. After thresholding, fragments can be culled accordingly (third row), or mapped back to an object ID in order to apply the corresponding transfer function (fourth row).

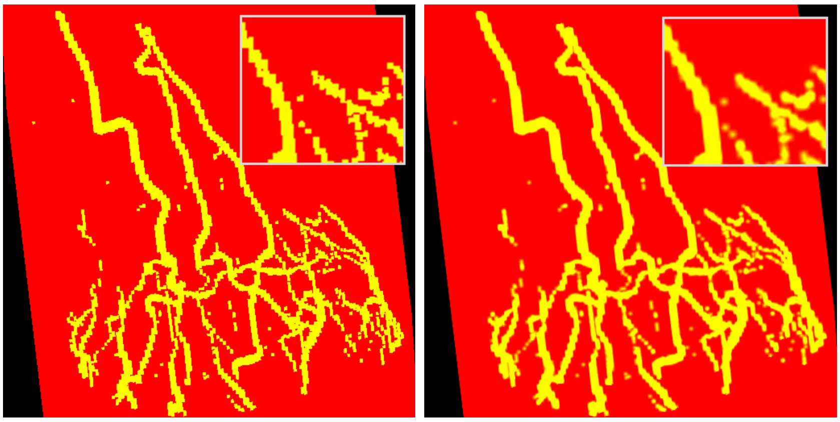


Figure 4.9: Selecting the transfer function on a per-fragment basis. In the left image, point-sampling of the object ID volume has been used, whereas in the right image procedural linear interpolation in the fragment shader achieves results of much better quality.

Multiple per-object transfer functions in a single rendering pass

In addition to simply determining whether a given fragment belongs to a currently active object or not, which has been described in the previous section, this filtering approach can be extended to the application of multiple transfer functions in a single rendering pass without sacrificing filtering quality. Figure 4.9 shows the difference in quality for two objects with different transfer functions (one entirely red, the other entirely yellow for illustration purposes).

In general hardware-accelerated volume rendering, the easiest way to apply multiple transfer functions in a single rendering pass would be to use the original volume texture with linear interpolation, and an additional separate point-sampled object ID texture. Although actual volume and ID textures could be combined into a single texture, the use of a separate texture to store the IDs is mandatory in order to prevent that filtering of the actual volume data also reverts back to point-sampling, since a single texture cannot use different filtering modes for different channels and point-sampling is mandatory for the ID texture. The hardware-native linear interpolation cannot be turned on in order to filter object IDs, and thus the resolution of the ID volume is easily discernible if the transfer functions are sufficiently different.

In order to avoid the artifacts related to point-sampling the ID texture, we perform several almost identical filtering steps in the fragment shader, where each of these steps simultaneously filters the object boundaries of four different objects. After the fragment's object ID has been determined via filtering, it can be used to access the global transfer function table as described in section 4.2 and illustrated in figure 4.6. For multiple simultaneous transfer functions, we do not use object set membership textures but the similar extended concept of *transfer function assignment textures*, which is illustrated in the right image of figure 4.5.

Each of these textures can be used for filtering the object ID volume with respect to four different object IDs at the same time by using the four channels of an RGBA texture in order to perform four simultaneous binary classification operations. In order to create these textures, each object set membership texture is converted into $\lceil \#objects/4 \rceil$ transfer function assignment textures, where $\#objects$ denotes the number of objects with different transfer functions in a given object set. All values of 1.0 corresponding to the first transfer function are stored into the red channel of this texture, those corresponding to the second transfer function into the green channel, and so on.

In the fragment shader, bi-linear interpolation must index this texture at four different locations given by the object IDs of the four input values to interpolate. This classifies the four input object IDs with respect to four objects with just four 1D texture sampling operations. A single linear interpolation step yields the linear interpolation of these four object classifications, which can then be compared against a threshold of $(0.5, 0.5, 0.5, 0.5)$, also requiring only a single operation for four objects. Interpolation and thresholding yields a vector with at most one component of 1.0, the other components set to 0.0. In order for this to be true, we require that interpolated and thresholded repeated binary classifications never overlap, which is not guaranteed for all types of filter kernels. In the case of bi-linear or tri-linear interpolation, however, overlaps can never occur [154].

The final step that has to be performed is mapping the binary classification to the desired object ID. We do this via a single dot product with a vector containing the four object IDs corresponding to the four channels of the transfer function assignment texture (figure 4.5, right). By calculating this dot product, we multiply exactly the object ID that should be assigned to the final fragment by 1.0. The other object IDs are multiplied by 0.0 and thus do not change the result. If the result of the dot product is 0.0, the fragment does not belong to any of the objects under consideration and can be culled. Note that exactly for this reason, we do not use object IDs of zero.

For the application of more than four transfer functions in a single rendering pass, the steps outlined above can be executed multiple times in the fragment shader. The results of the individual dot products are simply summed up, once again yielding the ID of the object that the current fragment belongs to.

Note that the calculation of filter weights is only required once, irrespective of the number of simultaneous transfer functions, which is also true for sampling the original object ID textures.

Equation 4.1 gives the major fragment shader resource requirements of our filtering and binary classification approach for the case of bi-linear interpolation with `LRP` instructions:

$$4\mathbf{TEX\_2D} + 4\left\lceil \frac{\#objects}{4} \right\rceil \mathbf{TEX\_1D} + 3\left\lceil \frac{\#objects}{4} \right\rceil \mathbf{LRP}, \tag{4.1}$$

in addition to one dot product and one thresholding operation (e.g., `DP4` and `SGE` instructions, respectively) for every $\lceil \#objects/4 \rceil$ transfer functions evaluated in a single pass.

Similarly to the alternative linear interpolation using texture-based filtering that we have outlined in section 4.3, procedural weight calculation and the `LRP` instructions can once again also be substituted by texture fetches and a few cheaper ALU instructions. On the Radeon 9700, we are currently able to combine high-quality shading with up to eight transfer functions in the same fragment shader, i.e., we are using up to two transfer function assignment textures in a single rendering pass.

4.4 Two-Level Volume Rendering

The final component of the framework presented in this chapter with respect to the separation of different objects is the possibility to use individual object-local compositing modes, as well as a single global compositing mode, i.e., *two-level volume rendering* [43, 44]. The

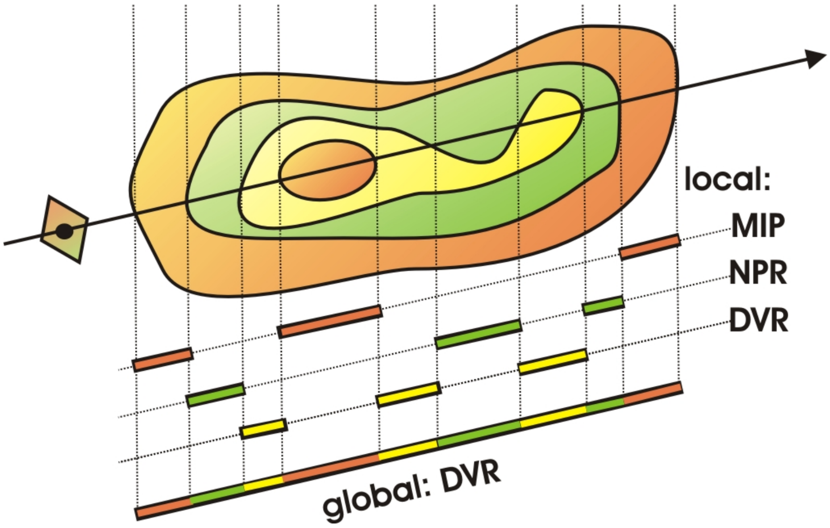


Figure 4.10: A single ray corresponding to a given image pixel is allowed to pierce objects that use their own object-local compositing mode. The contributions of different objects along a ray are combined with a single global compositing mode. Rendering a segmented data set with these two conceptual levels of compositing (local and global) is known as *two-level volume rendering*.

local compositing modes that can currently be selected are alpha blending (e.g., for DVR or tone shading), maximum intensity projection (e.g., for MIP or contour enhancement), and isosurface rendering. Global compositing can either be done by alpha blending, MIP, or a simple summation of all contributions.

Although the basic concept of two-level volume rendering is best explained using an image-order approach, i.e., individual rays (figure 4.10), in the context of texture-based volume rendering we have to implement it in object-order. As described in section 4.2, we are using two separate rendering buffers, a local and a global compositing buffer, respectively. Actual volume slices are only rendered into the local buffer, using the appropriate local compositing mode. When a new fragment has a different local compositing mode than the pixel that is

```
TransferLocalBufferIntoGlobalBuffer() {
  ActivateContextGlobalBuffer();
  DepthTest( NOT_EQUAL );
  StencilTest( RENDER_ALWAYS, SET_ONE );
  RenderSliceCompositingIds( DEPTH_BUFFER );
  DepthTest( DISABLE );
  StencilTest( RENDER_WHERE_ONE, SET_ZERO );
  RenderLocalBufferImage( COLOR_BUFFER );
}
```

Table 4.2: Detecting for all pixels simultaneously where the compositing mode changes from one slice to the next, and transferring those pixels from the local into the global compositing buffer.

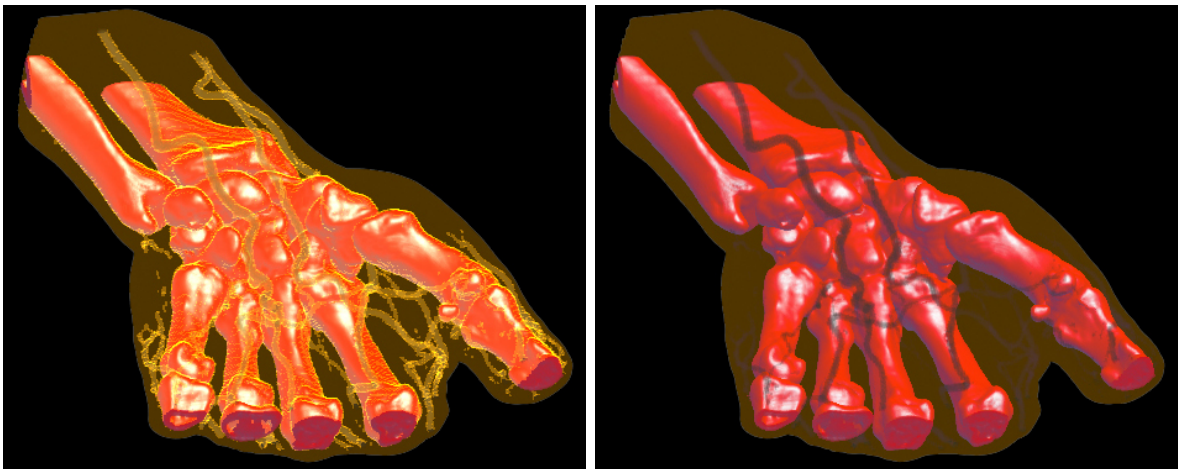


Figure 4.11: Detecting changes in compositing mode for each individual sample along a ray can be done exactly using two rendering buffers (left), or approximately using only a single buffer (right).

currently stored in the local buffer, that pixel has to be transferred into the global buffer using the global compositing mode. Afterward, these transferred pixels have to be cleared in the local buffer before the corresponding new fragment is rendered. Naturally, it is important that both the detection of a change in compositing mode and the transfer and clear of pixels is done for all pixels simultaneously.

In order to do this, we are using the depth buffer of both the local and the global compositing buffer to track the current local compositing mode of each pixel, and the stencil buffer to selectively enable pixels where the mode changes from one slice to the next. Before actually rendering a slice (see table 4.1), we render IDs corresponding to the local compositing mode into both the local and the global buffer's depth buffer. During these passes, the stencil buffer is set to one where the ID already stored in the depth buffer (from previous passes) differs from the ID that is currently being rendered. This gives us both an updated ID image in the depth buffer, and a stencil buffer that identifies exactly those pixels where a change in compositing mode has been detected.

We then render the image of the local buffer into the global buffer. Due to the stencil test, pixels will only be rendered where the compositing mode has actually changed. Table 4.2 gives pseudo code for what is happening in the global buffer. Clearing the just transferred pixels in the local buffer works almost identically. The only difference is that in this case we do not render the image of another buffer, but simply a quad with all pixels set to zero. Due to the stencil test, pixels will only be cleared where the compositing mode has actually changed.

Note that all these additional rendering passes are much faster than the passes actually rendering and shading volume slices. They are independent of the number of objects and use extremely simple fragment shaders. However, the buffer/context switching overhead is quite noticeable, and thus correct separation of compositing modes can be turned off during interaction. Figure 4.11 shows a comparison between approximate and correct compositing with one and two compositing buffers, respectively. Performance numbers can be found in table 4.3. When only a single buffer is used, the compositing mode is simply switched according to each new fragment without avoiding interference with the previous contents of the frame buffer.

The visual difference depends highly on the combination of compositing modes and spatial locations of objects. The example in figure 4.11 uses MIP and DVR compositing in order to highlight the potential differences. However, using approximate compositing is very useful for faster rendering, and often exhibits little or no loss in quality. Also, it is possible to get an almost seamless performance/quality trade-off between the two, by performing the buffer transfer only every n slices instead of every slice. See figures 4.12, 4.13, and 4.14 for two-level volume renderings of segmented volume data.

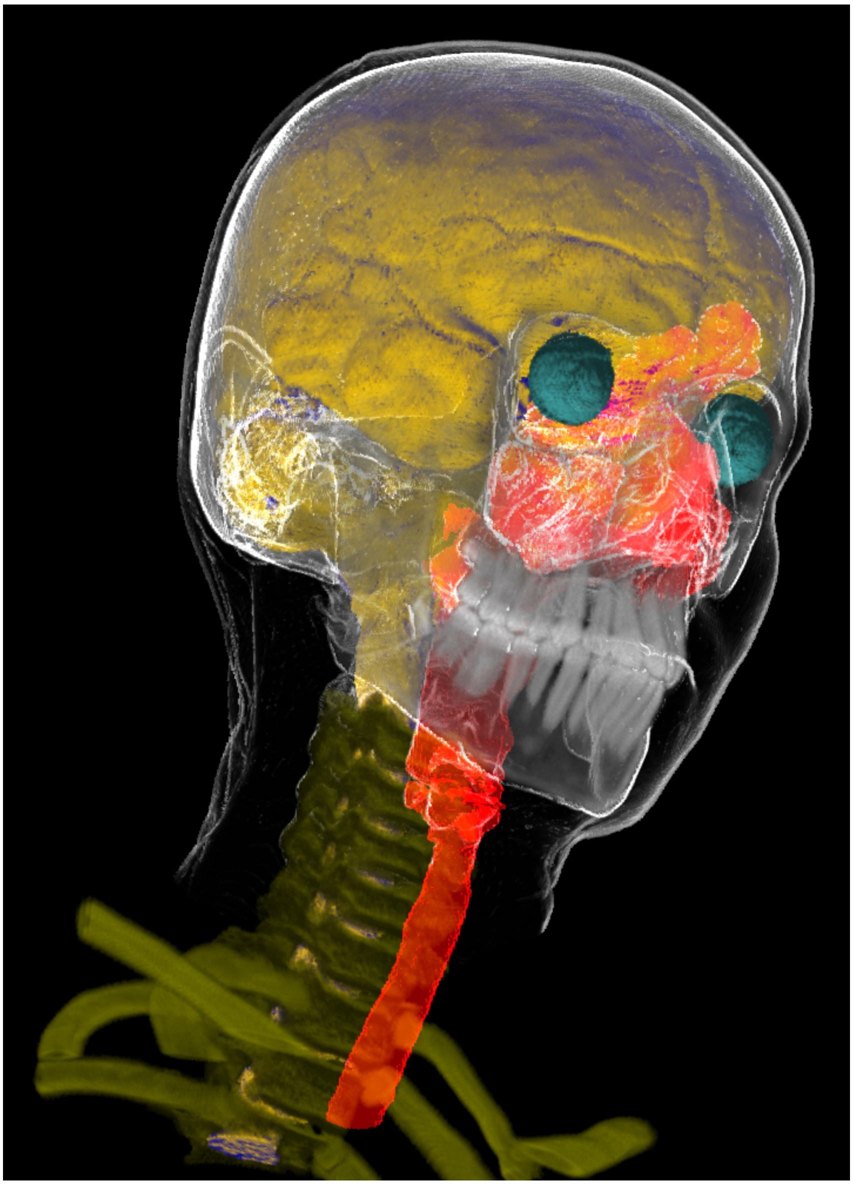


Figure 4.12: Segmented head and neck data set (256x256x333) with eight different enabled objects – brain: tone shading; skin: contour enhancement with clipping plane; eyes and spine: shaded DVR; skull, teeth, and vertebrae: unshaded DVR; trachea: MIP.

4.5 Performance

Actual rendering performance depends on a lot of different factors, so table 4.3 shows only some example figures. In order to concentrate on performance of rendering segmented data, all rates have been measured with unshaded DVR. Slices were object-aligned; objects were rendered all in a single pass (*single*) or in one pass per object (*multi+ztest*).

Compositing performance is independent of the rendering mode, i.e., can also be measured with DVR for all objects. Frame rates in parentheses are with linear boundary filtering enabled, other rates are for point-sampling during interaction. Note that in the unfiltered case with a single rendering pass for all objects, the performance is independent of the number of objects.

If more complex fragment shaders than unshaded DVR are used, the relative performance speed-up of *multi+ztest* versus *multi* increases further toward *single* performance, i.e., the additional overhead of writing object set IDs into the depth buffer becomes negligible.

#slices	#objects	compositing	single [fps]	multi+ztest [fps]	*multi* [fps]
128	3	one buffer	48 (16.2)	29.2 (15.4)	*19.3 (6.8)*
128	3	two buffers	7 (3.9)	6.2 (3.2)	*5 (1.9)*
128	8	one buffer	48 (11.3)	15.5 (10)	*7 (2.1)*
128	8	two buffers	7 (3.2)	5.4 (3)	*2.5 (0.7)*
256	3	one buffer	29 (9.1)	15.6 (8.2)	*11 (3.4)*
256	3	two buffers	3.5 (2)	3.2 (1.8)	*2.5 (1.1)*
256	8	one buffer	29 (5.3)	8.2 (5.2)	*3.7 (1.1)*
256	8	two buffers	3.5 (1.7)	3.1 (1.6)	*1.2 (0.4)*

Table 4.3: Performance on an ATI Radeon 9700; 512x512 viewport size; 256x128x256 data set; three and eight enabled objects, respectively. Numbers are in frames per second. Compositing is done with either one or two buffers, respectively. The *multi* column with early z-testing turned off is only shown for comparison purposes in order to illustrate the impact of rendering and shading the entire volume multiple times when the early z-test is not employed.

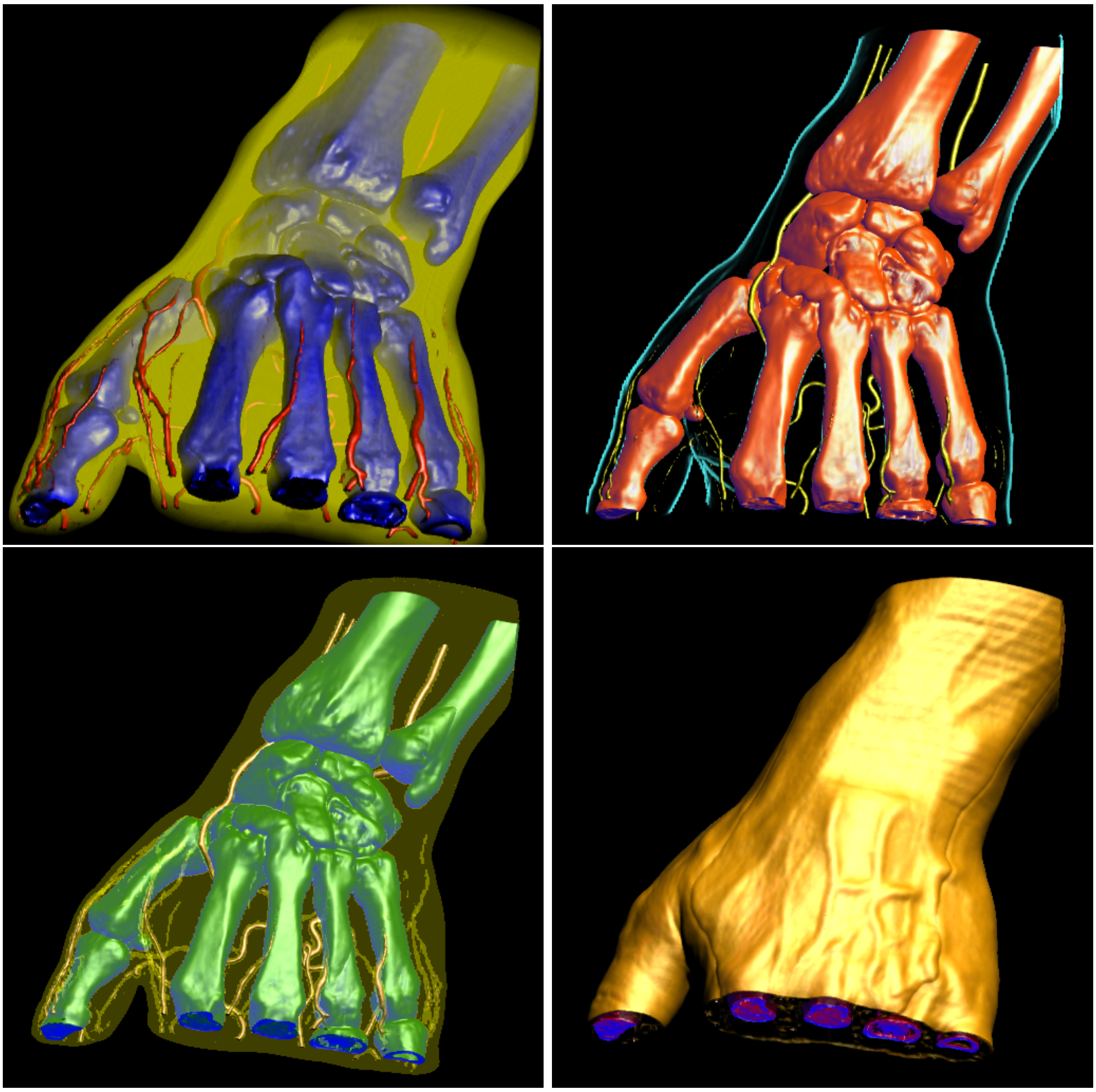


Figure 4.13: Hand data set (256x128x256) examples of different rendering and compositing modes. (top, left) skin with unshaded DVR, vessels and bones with shaded DVR; (top, right) skin with contour rendering, vessels with shaded DVR, bones with tone shading; (bottom, left) skin with MIP, vessels with shaded DVR, bones with tone shading; (bottom, right) skin with isosurfacing, occluded vessels and bones with shaded DVR.

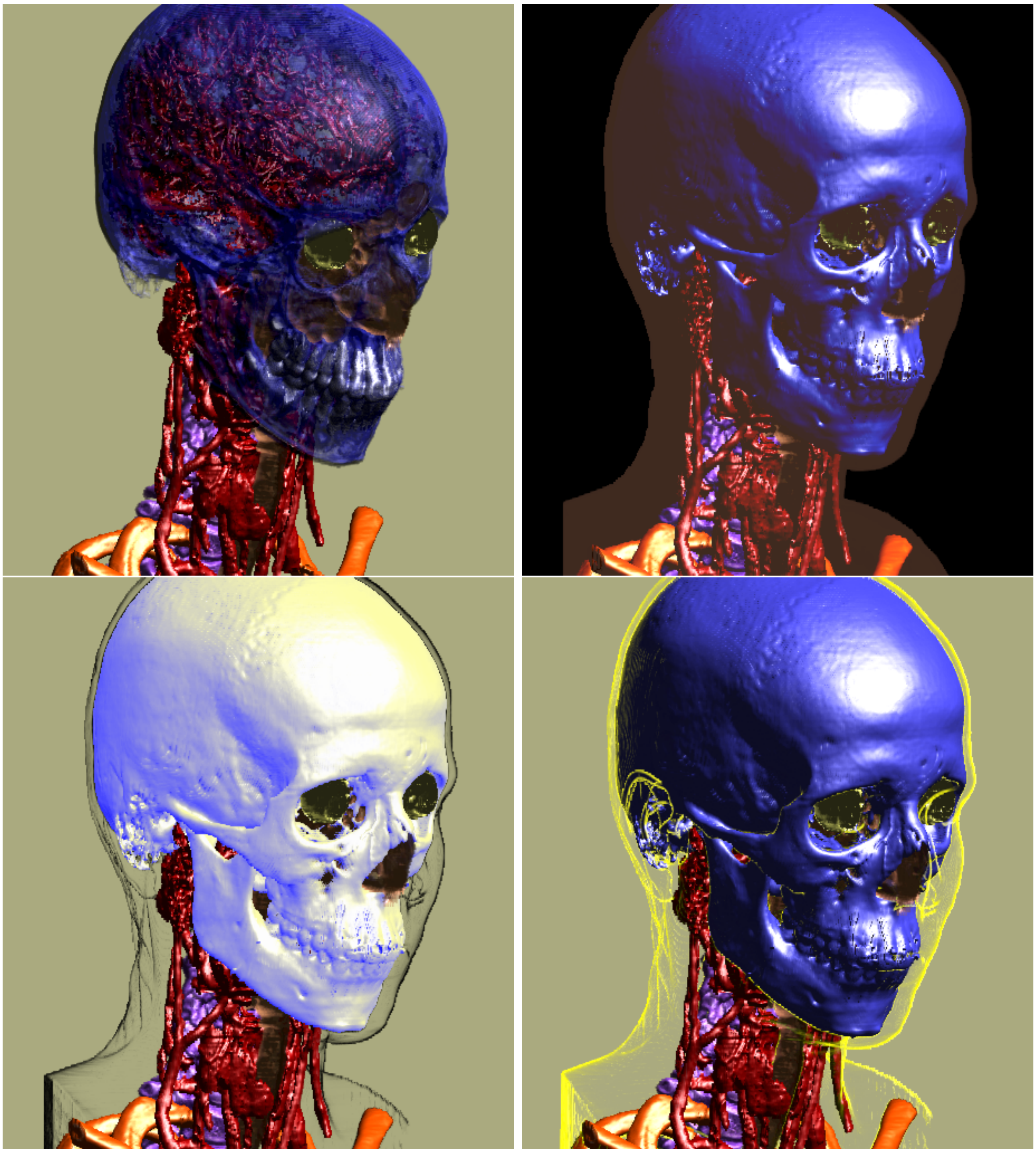


Figure 4.14: Head and neck data set (256x256x333) examples of different rendering and compositing modes. (top, left) skin disabled, skull with shaded DVR; (top, right) skin with MIP, skull with isosurfacing; (bottom, left) skin with contour rendering, skull with tone shading; (bottom, right) skin with contour rendering, skull with isosurfacing.

Chapter 5

Non-Photorealistic Volume Rendering

Parts of this chapter are based on the technical report *Real-Time High-Quality Rendering of Isosurfaces* [144].

Non-photorealistic rendering techniques, e.g., rendering styles imitating artistic illustration, have established themselves as a very powerful tool for conveying a specific meaning in rendered images, especially in renderings of surfaces [29, 146]. In recent years, the interest in adapting existing NPR techniques to volumes and creating new entirely volumetric NPR models has increased significantly [16, 22, 99].

This chapter first shows two examples of integrating simple non-photorealistic techniques into real-time volume rendering by evaluating the corresponding shading equations directly in the hardware fragment shader. It then focuses on real-time rendering of isosurfaces with more sophisticated NPR techniques based on implicit curvature information, which has previously been demonstrated for off-line volume rendering [67].

A very important basic concept introduced in section 5.3 is the notion of *deferred shading.* When deferred shading is employed, the actual data specified in object space, i.e., in our case the volume, are first processed without any potentially expensive shading computations in order to determine actually visible pixels. Shading is then performed in subsequent rendering passes in image space for these pixels only.

We demonstrate how the concept of deferred shading can be used for high-quality shading of isosurfaces, as well as deferred computation of differential implicit surface properties such as partial first and second order derivatives and principal curvatures. An important property of these computations is that we employ tri-cubic filtering with a cubic B-spline filter and its first and second derivatives in order to attain high-quality results.

5.1 Basic Non-Photorealistic Rendering Modes

This section outlines two simple examples of non-photorealistic volume rendering modes. First, the concept of shading surfaces with tone shading [28] can be adapted to volume shading by incorporating color and opacity retrieved from the transfer function. Second, we outline a simple model for rendering the silhouettes of material boundaries in volumes that does not use an explicit notion of surfaces but modulates a contour intensity depending on the angle between view and gradient direction by the gradient magnitude [16].

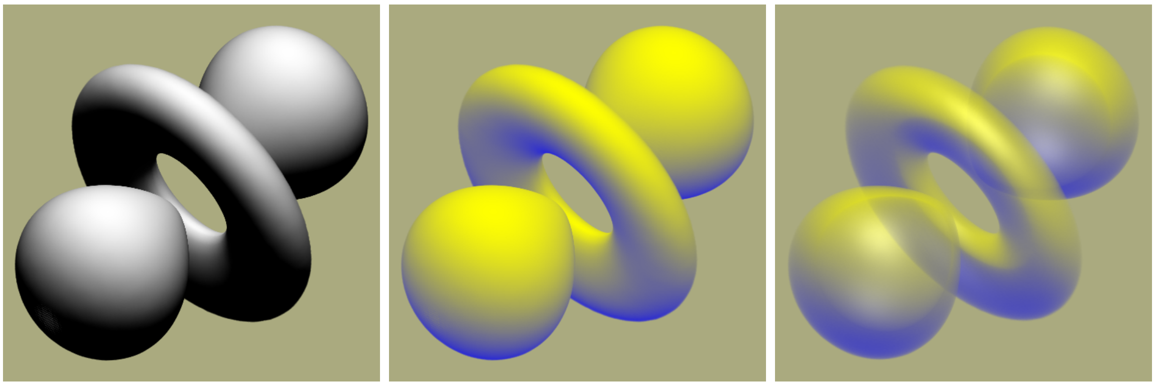


Figure 5.1: Comparison of standard volume shading (left) and tone shading (center) with an isosurface-like transfer function. Incorporating opacity from a transfer function reveals the volumetric structure of the rendered model (right).

Tone shading

In contrast to Blinn-Phong shading, which determines a single light intensity depending on the dot product between the view vector and the surface normal, tone shading [28] (sometimes also called Gooch shading) interpolates between two user-specified colors over the full $[-1,1]$ range of this dot product. Traditionally, one of these colors is set to a warm tone, e.g., orange or yellow, and the other one to a cool tone, e.g., purple or blue. Cool colors are perceived by human observers as receding into the background, whereas warm colors are seen as being closer to the foreground [28]. Tone shading uses this observation to improve depth perception of shaded images.

Although originally developed for surface shading, tone shading can easily be adapted to direct volume rendering by mixing the color from the transfer function with the color obtained via tone shading. One of the possibilities to do this is the following:

$$\mathbf{I} = \left(\frac{1+\mathbf{l}\cdot\mathbf{n}}{2}\right)k_a + \left(1-\frac{1+\mathbf{l}\cdot\mathbf{n}}{2}\right)k_b, \tag{5.1}$$

where $\mathbf{l}$ denotes the light vector, and $\mathbf{n} = \nabla f/|\nabla f|$ is the normalized gradient of the scalar field f that is used as normal vector.

The two colors to interpolate, k_a and k_b, are derived from two constant colors k_{cool} and k_{warm} and the color from the transfer function k_t, using two user-specified factors α and β that determine the additive contribution of k_t:

$$k_a = k_{cool} + \alpha k_t \tag{5.2}$$

$$k_b = k_{warm} + \beta k_t \tag{5.3}$$

The opacity of the shaded fragment is determined directly from the transfer function lookup, i.e., the alpha portion of k_t.

These tone shading equations can easily be evaluated in the hardware fragment shader on a per-fragment basis for high-quality results. Figure 5.1 shows example images.

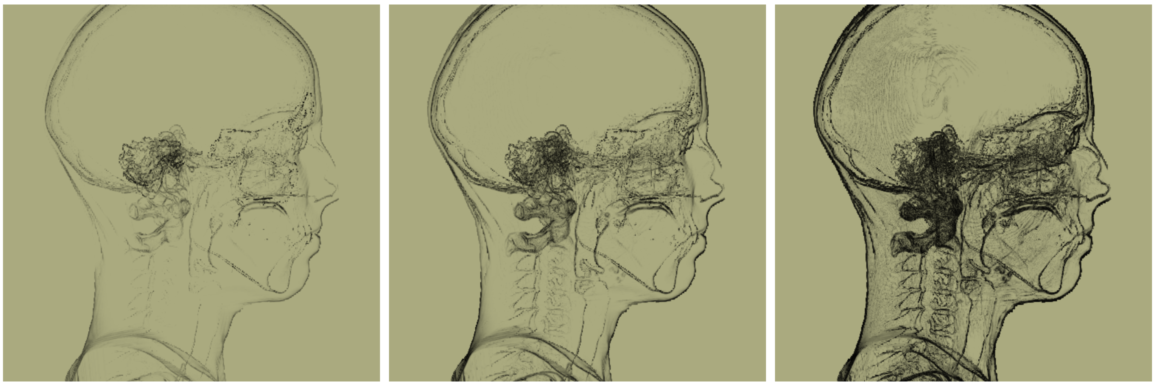


Figure 5.2: Simple contour enhancement based on gradient magnitude and angle between view and local gradient direction. The gradient magnitude windowing function $g(\cdot)$ is an easy way to control contour appearance. These images simply use three different window settings.

Contour enhancement

Even without an explicit notion of surfaces, or isosurfaces, a very simple model based on gradient magnitude and the angle between the view and gradient direction can visualize the silhouettes of material boundaries in volumes quite effectively [16]. This model can be used in real-time volume rendering for obtaining a contour intensity I by procedural evaluation of the following equation in the hardware fragment shader:

$$\mathbf{I} = g\big(|\nabla f|\big) \cdot \big(1 - |\mathbf{v} \cdot \mathbf{n}|\big)^8, \tag{5.4}$$

where $\mathbf{v}$ is the viewing vector, ∇f denotes the gradient of a given voxel, $\mathbf{n} = \nabla f / |\nabla f|$ is the normalized gradient, and $g(\cdot)$ is a windowing function for the gradient magnitude.

The windowing function $g(\cdot)$ is illustrated in figure 5.3, and figure 5.2 shows three example results of using different window settings. The window can be specified directly via its center and width. Alternatively, it can also be specified through a standard transfer function interface, where the alpha component is the weighting factor for the view-dependent part, and the RGB components are simply neglected.

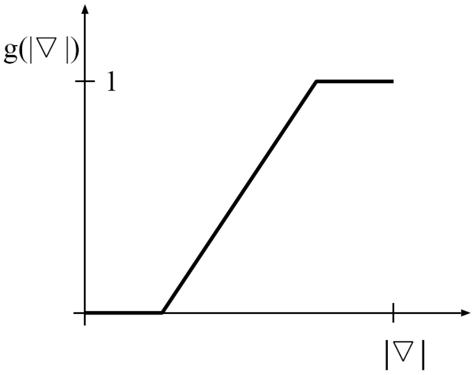


Figure 5.3: Windowing of gradient magnitude [16] in order to restrict the detection of contours to the interfaces, i.e., boundary surfaces, between different materials.

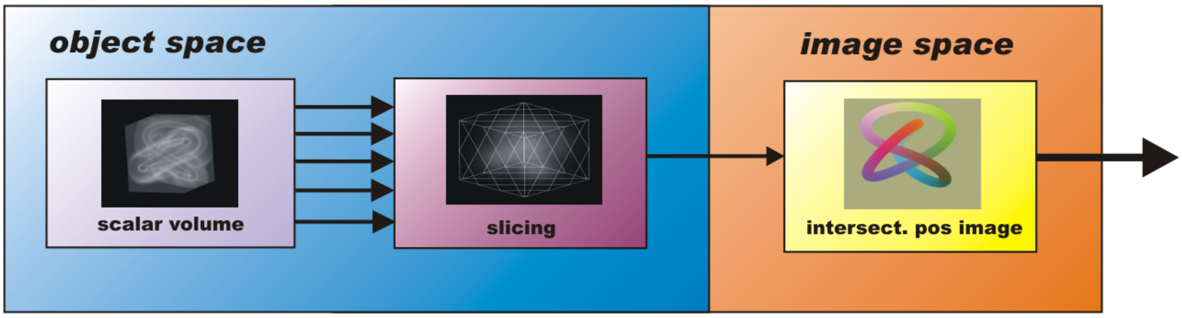


Figure 5.4: Slicing a volume in order to determine an intersection image of ray-isosurface intersections for deferred shading of an isosurface in subsequent image space rendering passes.

The obtained fragment intensity I can be multiplied by a constant contour color in order to render colored contours. If alpha blending is used as compositing mode, the fragment alpha can simply be set to the intensity I. However, a very useful compositing mode for contours obtained via this technique is maximum intensity projection (MIP), instead of using alpha blending.

Combination with segmented data

Using the two non-photorealistic rendering modes that have been outlined above for rendering segmented data with per-object rendering modes is a very powerful approach to emphasizing specific object structures in volume data.

Rendering of contours, for example, is a good way to provide context for focus regions rendered with more traditional volume rendering techniques. Tone shading is naturally suited as shading mode for rendering isosurfaces or structures with high opacity, whereas objects rendered with lower opacity could be rendered with standard direct volume rendering, for example.

See chapter 4 for examples of combining traditional and non-photorealistic techniques in a single volume rendering in order to enhance perception of individual objects of interest and separate context from focus regions.

5.2 Isosurfaces

Many non-photorealistic volume rendering techniques operate on isosurfaces of volumetric data. Although direct volume rendering as well as other techniques aiming to depict an entire volume in a single image are very important and popular, rendering isosurfaces corresponding to particular structures of interest, or more precisely, their boundaries, play a very important role in the field of volume rendering.

There are two major approaches for rendering isosurfaces of volume data. First, an explicit triangle mesh corresponding to a given iso-value can be extracted prior to rendering, e.g., using marching cubes [97] or one of its variants [74]. Second, ray-isosurface intersections can be determined via ray casting [3, 89]. Naturally, general NPR techniques for rendering surfaces can easily be applied to rendering isosurfaces of volume data.

In hardware-accelerated volume rendering, isosurfaces have traditionally been rendered by slicing the volume in back-to-front order and exploiting the hardware alpha test in order

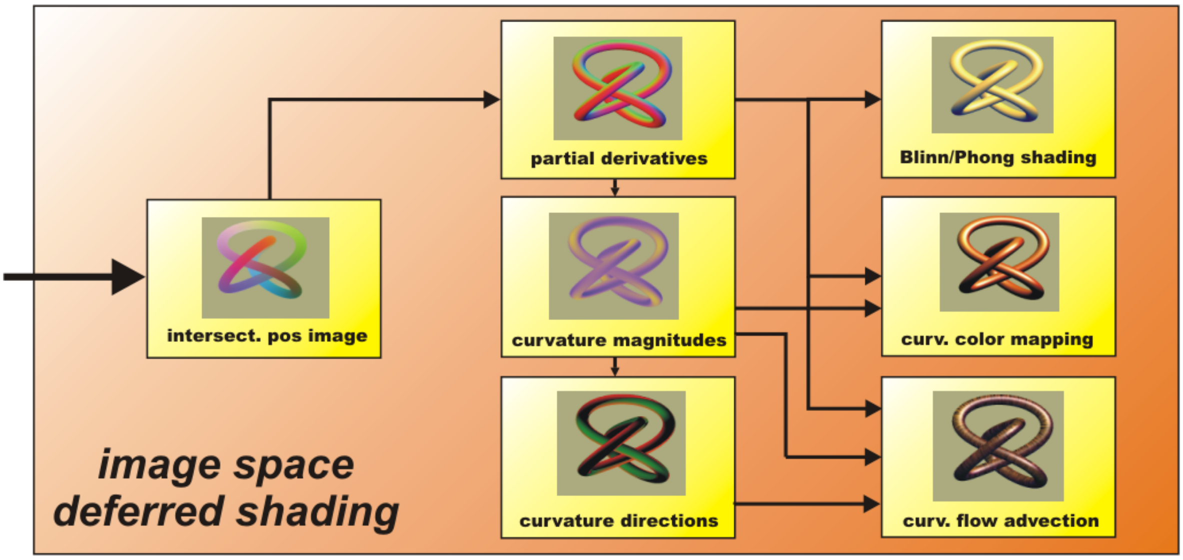


Figure 5.5: Deferred shading computations for an isosurface given as a floating point image of ray-surface intersection positions. First, differential properties such as the gradient and additional partial derivatives can be computed. These derivatives also allow to compute principal curvature information on-the-fly. In the final shading pass, the obtained properties can be used for high-quality shading computations. All of these computations and shading operations have image space instead of object space complexity and are only performed for visible pixels.

to reject fragments not corresponding to the isosurface [174]. The concept of pre-integration can also be applied to isosurface rendering, which yields results of high quality even with low sampling rates [24]. Recently, GPU-based ray casting approaches have been developed [78, 137], which can also be used to determine ray-isosurface intersections.

The following sections illustrate a high-quality rendering pipeline for direct rendering of isosurfaces by determining ray-isosurface intersections and subsequent deferred shading of the corresponding pixels. The input to the deferred shading stages is a floating point image of ray-isosurface intersection positions, which is obtained from either slicing the volume [34], illustrated in figure 5.4, or first hit ray casting that stores hit positions into the target buffer using an adaptation of a GPU ray casting method presented recently [78].

5.3 Deferred Shading

In standard rendering pipelines, shading equations are often evaluated for pixels that are entirely invisible or whose contribution to the final image is negligible. With the shading equations used in real-time rendering becoming more and more complex, avoiding these computations for invisible pixels becomes an important goal.

A very powerful concept that allows to compute shading only for actually visible pixels is the notion of *deferred shading*. Deferred shading computations are usually driven by one or more input images that contain all the information that is necessary for performing the final shading of the corresponding pixels. The major advantage of deferred computations is that it reduces their complexity from being proportional to object space, e.g., the number of voxels in a volume, to the complexity of image space, i.e., the number of pixels in the final output image. Naturally, these computations are not limited to shading equations per se, but can also include the derivation of additional information that is only needed for visible pixels and may be required as input for shading, such as differential surface properties.

In this section, we describe deferred shading computations for rendering isosurfaces of volumetric data. The computations that are deferred to image space are not limited to actual shading, but also include the derivation of differential implicit surface properties such as the gradient (first order partial derivatives), the Hessian matrix (second order partial derivatives), and principal curvature information.

Figure 5.5 illustrates a pipeline for deferred shading of isosurfaces of volume data, and figure 5.6 shows example images corresponding to the output of specific image space rendering passes. The input to the pipeline is a single floating point image storing ray-surface intersection positions of the viewing rays and the isosurface. This image is obtained via either slicing the volume, or first hit ray casting, as outlined above and illustrated in figure 5.4.

From this intersection position image, differential isosurface properties such as the gradient and additional partial derivatives such as the Hessian matrix can be computed first. This allows shading with high-quality gradients, as well as computation of high-quality principal curvature magnitude and direction information. Sections 5.4 and 5.5 describe high-quality

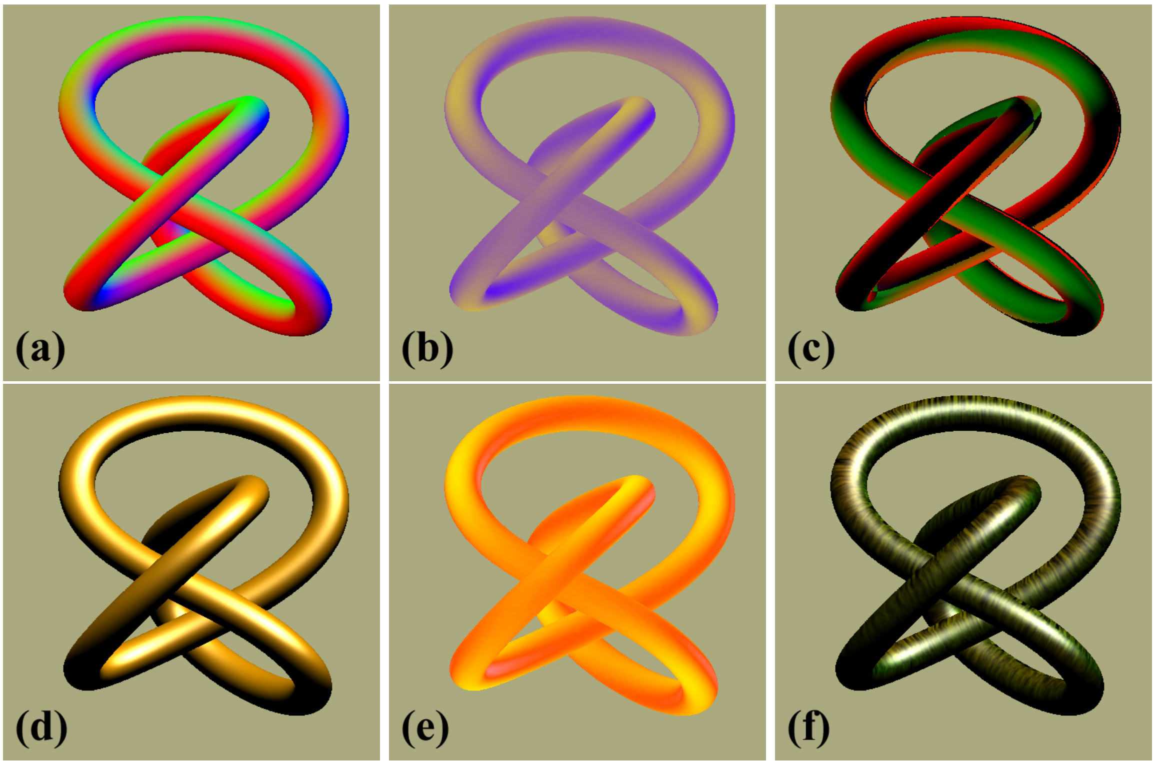


Figure 5.6: Example image space rendering passes of deferred isosurface shading. Surface properties such as (a) the gradient (here color-coded in RGB), (b) principal curvature magnitudes (here: κ_1), and (c) principal curvature directions can be reconstructed. These properties can be used in shading passes, e.g., (d) Blinn-Phong shading, (e) color coding of curvature measures (here: $\sqrt{\kappa_1^2 + \kappa_2^2}$ [67]), and (f) advection of flow along principal curvature directions.

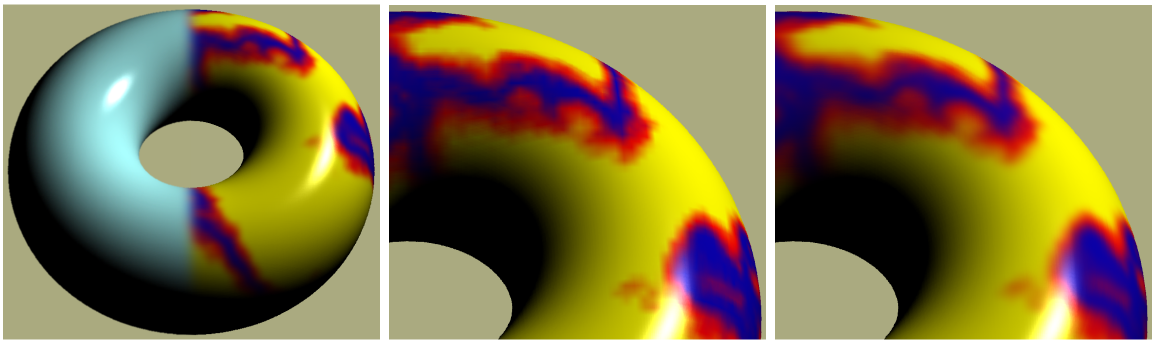


Figure 5.7: Deferred shading of an isosurface from a ray-isosurface intersection and a gradient image (left). Solid texture filtering can be done using tri-linear interpolation (center), or tri-cubic filtering in real-time (right).

reconstruction of differential isosurface properties.

In the final actual shading pass, differential surface properties can be used for shading computations such as Blinn-Phong shading, color mapping of curvature magnitudes, and flow advection along curvature directions, as well as applying a solid texture onto the isosurface.

Shading from gradient image

The simplest shading equations depend on the normal vector of the isosurface, i.e., its normalized gradient. The normal vector can for example be used to compute Blinn-Phong shading, and reflection and refraction mapping that index an environment map with vectors computed from the view vector and the normal vector. See figure 5.7(left) for an example.

Solid texturing

The initial position image that contains ray-isosurface intersection positions can be used for straight-forward application of a solid texture onto an isosurface. Parameterization is simply done by specifying the transformation of object space to texture space coordinates, e.g., via an affine transformation. For solid texturing, real-time tri-cubic filtering can be used instead of tri-linear interpolation in order to achieve high-quality results. See figure 5.7(center and right) for a comparison.

5.4 Deferred Gradient Reconstruction

The most important differential property of the isosurface that needs to be reconstructed is the gradient of the underlying scalar field f:

$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^{\mathbf{T}} \tag{5.5}$$

The gradient can then be used as implicit surface normal for shading and curvature computations.

The surface normal is the normalized gradient of the volume, or its negative, depending on the notion of being inside/outside of the object that is bounded by the isosurface: $\mathbf{n} = -\mathbf{g}/|\mathbf{g}|$. The calculated gradient can be stored in a single RGB floating point image, see figure 5.6(a).

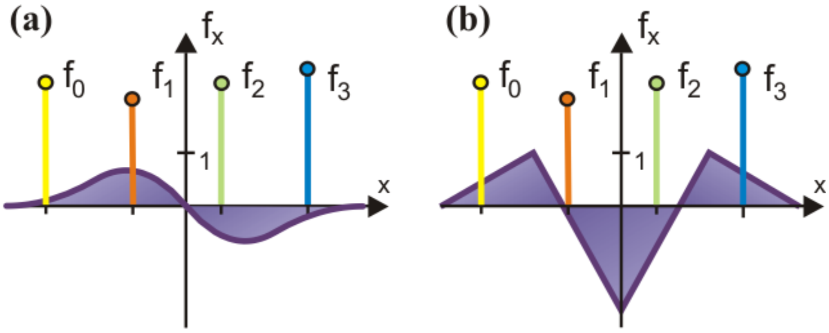


Figure 5.8: The first order (a) and second order (b) derivatives of the cubic B-spline filter for direct high-quality reconstruction of derivatives via convolution.

Hardware-accelerated high-quality filtering can be used for reconstruction of high-quality gradients by convolving the original scalar volume three times with the first derivative of a reconstruction kernel, e.g., the derived cubic B-spline kernel that is shown in figure 5.8(a).

The quality difference between cubic filtering and linear interpolation is even more apparent in gradient reconstruction than it is in value reconstruction. Figure 5.9 shows a comparison of different combinations of filters for value and gradient reconstruction, i.e., linear interpolation and cubic reconstruction with a cubic B-spline kernel. Figure 5.10 compares linear and cubic (B-spline) reconstruction using reflection mapping and a line pattern environment map. Reconstruction with the cubic B-spline achieves results with C^2 continuity.

5.5 Other Differential Properties

In addition to the gradient of the scalar volume, i.e., its first partial derivatives, further differential properties can be reconstructed in additional deferred shading passes.

For example, implicit principal curvature information can be computed from the second order partial derivatives of the volume. Curvature has many applications in surface investigation and rendering, e.g., non-photorealistic rendering equations incorporating curvature magnitudes in order to detect surface structures such as ridge and valley lines, or rendering

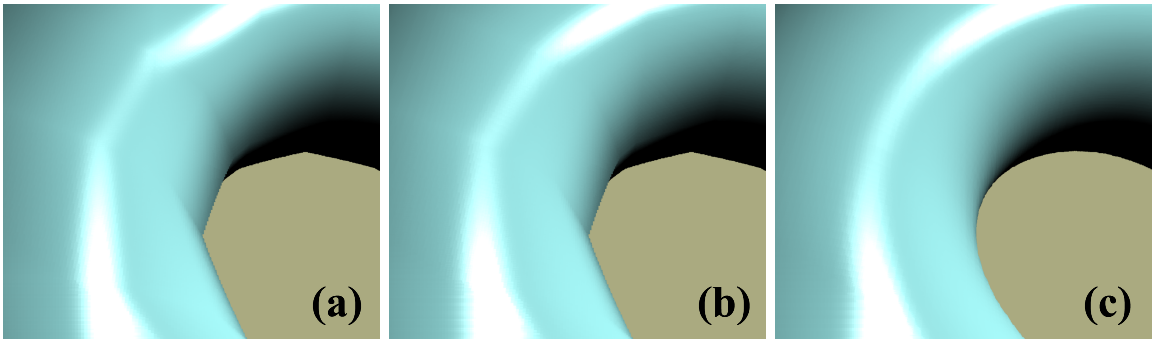


Figure 5.9: Linear and cubic filtering for value and gradient reconstruction on a torus: (a) both value and gradient are linear; (b) value is linear and gradient cubic; (c) both value and gradient are cubic. For cubic filtering, a cubic B-spline kernel has been used.

silhouettes of constant screen space thickness.

Second order partial derivatives: the Hessian

The Hessian matrix **H** is comprised of all second order partial derivatives of the scalar volume f:

$$\mathbf{H} = \nabla \mathbf{g} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{pmatrix} \tag{5.6}$$

Due to symmetry, only six unique components need to be calculated, which can be stored in two RGB floating point images.

High-quality second order partial derivatives can be computed by convolving the scalar volume with a combination of first and second order derivatives of the cubic B-spline kernel,

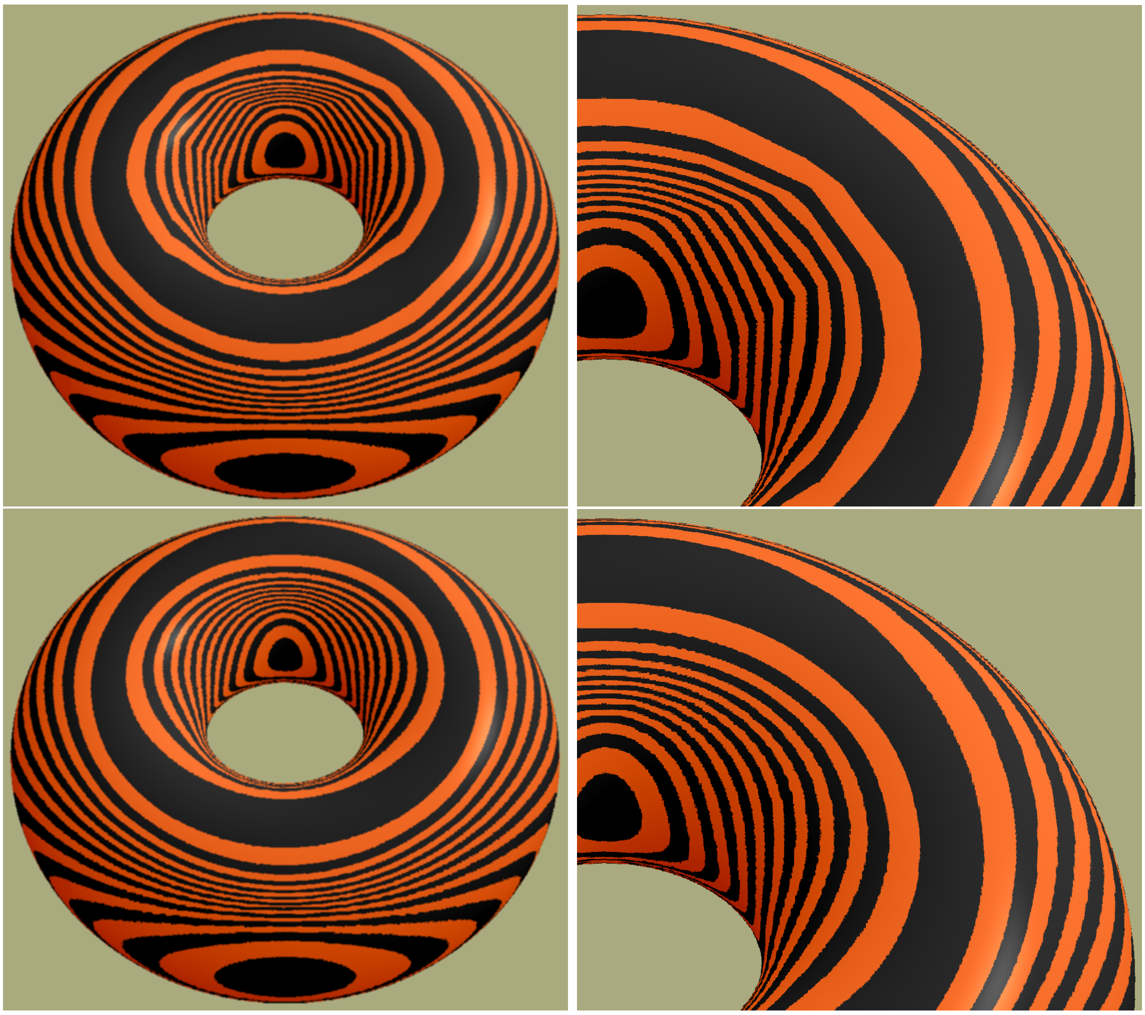


Figure 5.10: Comparing linear and cubic gradient reconstruction with a cubic B-spline filter using reflection lines. Top row images are with linear, bottom row with cubic filtering.

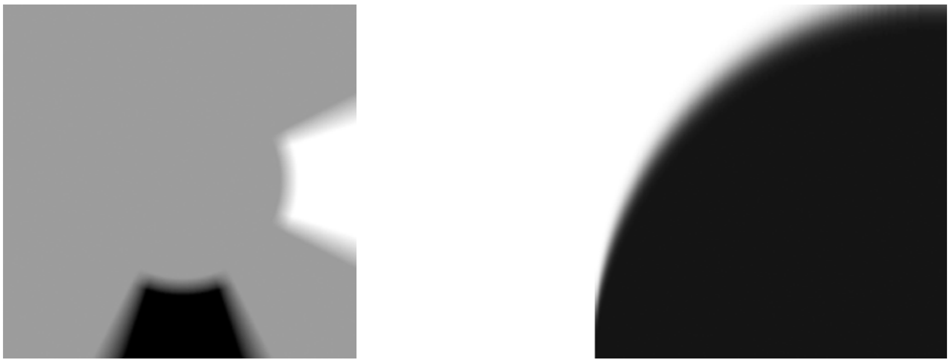


Figure 5.11: Two-dimensional transfer functions in curvature space. (left) Ridge and valley lines in (κ_1, κ_2) domain [67]; (right) contour thickness control [67].

for example, which is illustrated in figure 5.8.

Principal curvature magnitudes

The first and second principal curvature magnitudes (κ_1, κ_2) of the isosurface can be estimated directly from the gradient $\mathbf{g}$ and the Hessian $\mathbf{H}$ [67], whereby tri-cubic filtering in general yields high-quality results. This can be done in a single rendering pass, which uses the three partial derivative RGB floating point images generated by previous pipeline stages as input textures. It amounts to a moderate number of vector and matrix multiplications and solving a quadratic polynomial.

The result is a floating point image storing (κ_1, κ_2), which can then be used in the following passes for shading and optionally calculating curvature directions. See figure 5.6(b).

Principal curvature directions

The principal curvature magnitudes are the eigenvalues of a 2x2 eigensystem in the tangent plane specified by the normal vector, which can be solved in the next rendering pass for the corresponding eigenvectors, i.e., the 1D subspaces of principal curvature directions. Representative vectors for either the first or second principal directions can be computed in a single rendering pass.

The result is a floating point image storing principal curvature direction vectors. See Figure 5.6(c).

Filter kernel considerations

All curvature reconstructions in this chapter employ a cubic B-spline filter kernel and its derivatives. It has been shown that cubic filters are the lowest order reconstruction kernels for obtaining high-quality curvature estimates. They also perform very well when compared with filters of even higher order [67].

The B-Spline filter is a good choice for curvature reconstruction because it is the only fourth order BC-spline filter which is both accurate and continuous for first and second derivatives [111, 67]. Hence it is the only filter of this class which reconstructs continuous curvature estimates.

However, although B-spline filters produce smooth and visually pleasing results, they might be inappropriate in some applications where data interpolation is required [108]. Using a combination of the first and second derivatives of the cubic B-spline for derivative reconstruction, and a Catmull-Rom spline for value reconstruction is a viable alternative that avoids smoothing the original data [67].

5.6 Rendering from Implicit Curvature

Computing implicit surface curvature is a powerful tool for isosurface investigation and non-photorealistic rendering of isosurfaces.

When differential isosurface properties have been computed in preceding deferred shading passes (see section 5.5), this information can be used for performing a variety of mappings to shaded images in a final shading pass.

Curvature-based transfer functions

Principal curvature magnitudes can be visualized on an isosurface by mapping them to colors via one-dimensional or two-dimensional transfer function lookup textures.

One-dimensional curvature transfer functions. Simple color mappings of first or second principal curvature magnitude via 1D transfer function lookup tables can easily be computed during shading. The same approach can be used to depict additional curvature measures directly derived from the principal magnitudes, such as mean curvature $(\kappa_1 + \kappa_2)/2$ or Gaussian curvature $\kappa_1\kappa_2$. See figures 5.12(left), 5.16(top, left), and 5.18(top, left) for examples.

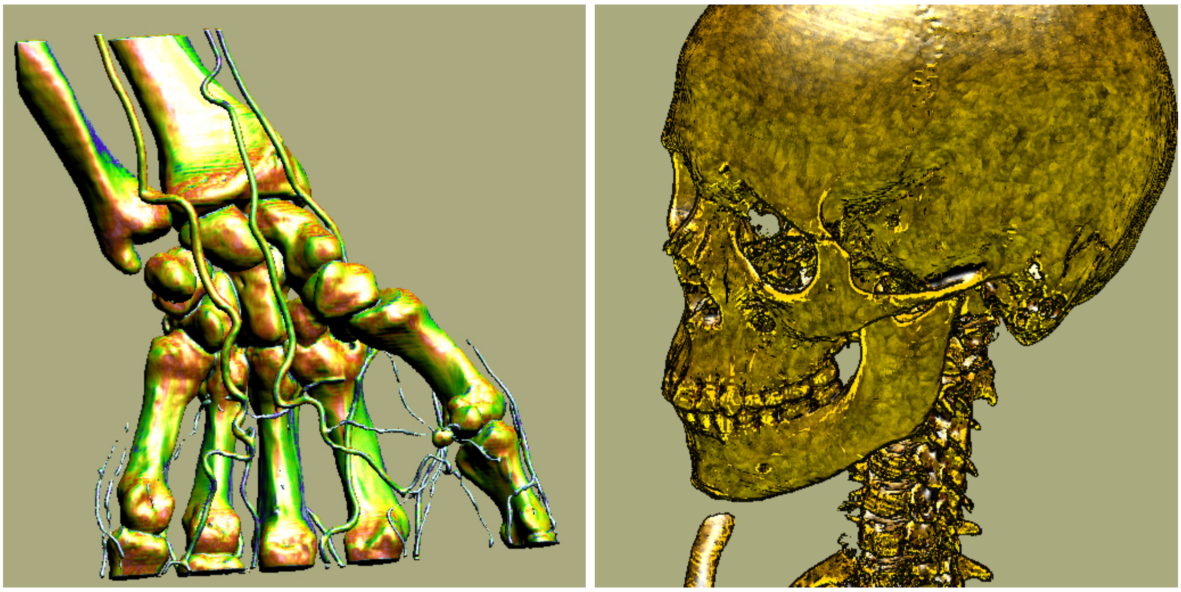


Figure 5.12: Two examples of implicit curvature-based isosurface rendering. (left) CT scan (256x128x256) with color mapping from a 1D transfer function depicting $\sqrt{\kappa_1^2 + \kappa_2^2}$; (right) CT scan (256x256x333) with contours, ridges and valleys, tone shading, and principal curvature-aligned flow advection to generate a noise pattern on the surface.

Two-dimensional curvature transfer functions. Transfer functions in the 2D domain of both principal curvature magnitudes (κ_1, κ_2) are especially powerful, since color specification in this domain allows to highlight different structures on the surface [50], including ridge and valley lines [57, 67]. Curvature magnitude information can also be used to implement silhouette outlining with constant screen space thickness [67]. See figures 5.12, 5.14, 5.16, 5.17, and 5.18 for examples. Figure 5.11 illustrates 2D transfer functions in the domain of curvature measures for ridge and valley lines (left), and constant silhouette thickness (right).

Curvature-aligned flow advection

Direct mappings of principle curvature directions to RGB colors are hard to interpret, see figure 5.6(c), for example.

However, principal curvature directions on an isosurface can be visualized using image-based flow visualization [162]. In particular, flow can be advected on the surface entirely in image space [82, 83]. These methods can easily be used in real-time, complementing the capability to generate high-quality curvature information on-the-fly, which also yields the underlying, potentially unsteady, "flow" field in real-time. See figure 5.6(f). In this case, it is natural to perform per-pixel advection guided by the floating point image containing principal direction vectors instead of warping mesh vertex or texture coordinates.

A problem with advecting flow along curvature directions is that their orientation is not uniquely defined and thus seams in the flow cannot be entirely avoided [162].

See figures 5.13 and 5.14(top, left) for examples.

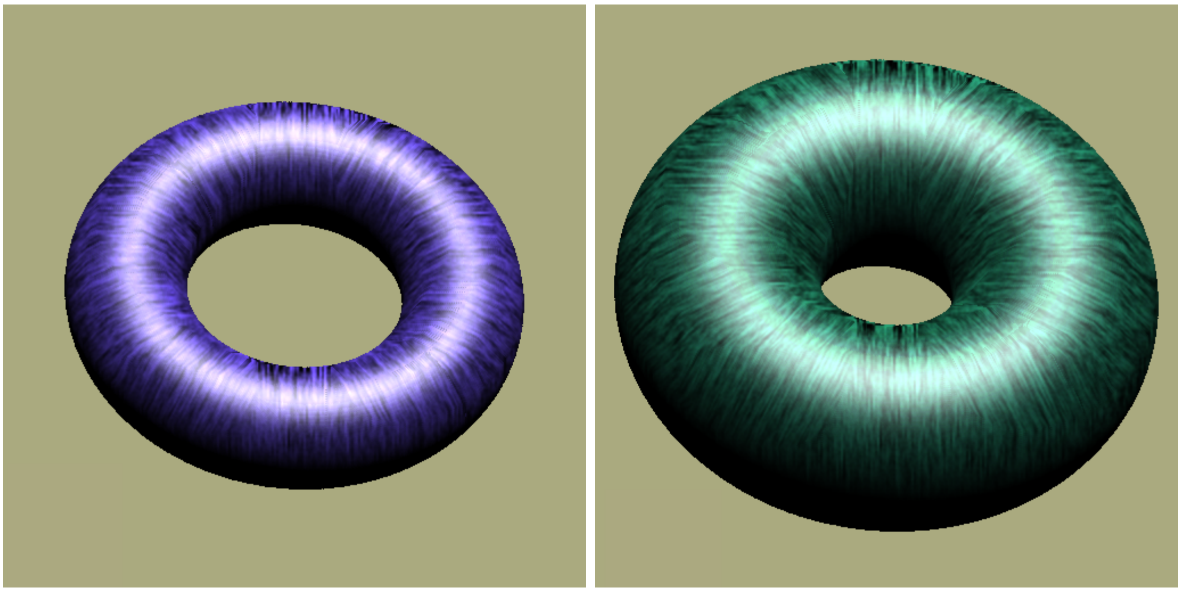


Figure 5.13: Changing the iso-value of a torus isosurface represented by a signed distance field. Color is derived from maximum principal curvature magnitude, and flow is advected in image space in maximum principal curvature direction. All changes to the iso-value take effect in real-time. Curvature directions constitute an unsteady flow field.

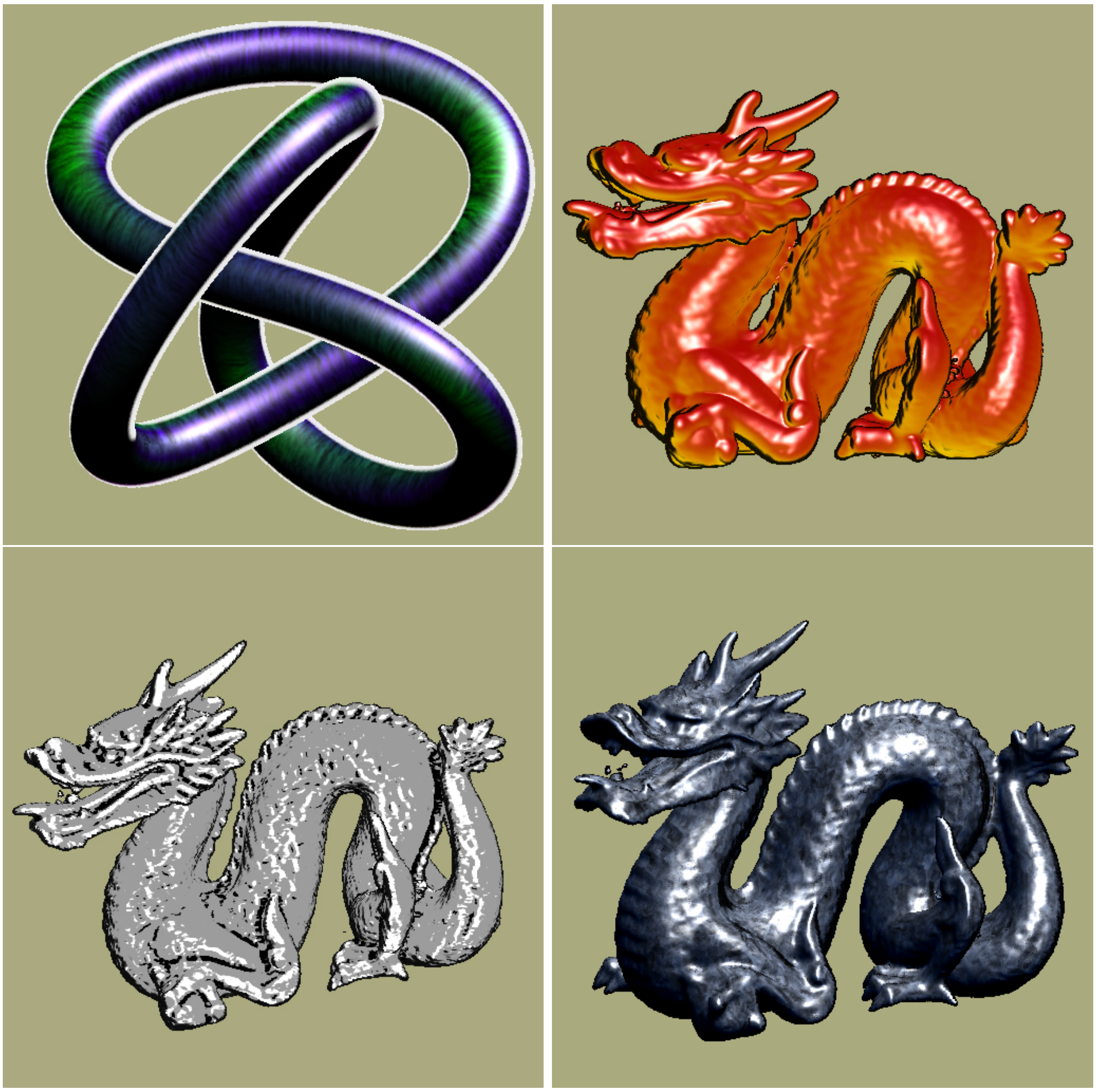


Figure 5.14: Curvature-based NPR. (top, left) Contours, curvature magnitude colors, and flow in curvature direction; (top, right) tone shading and contours; (bottom, left) contours, ridges, and valleys; (bottom, right) flow in curvature direction with Phong shading.

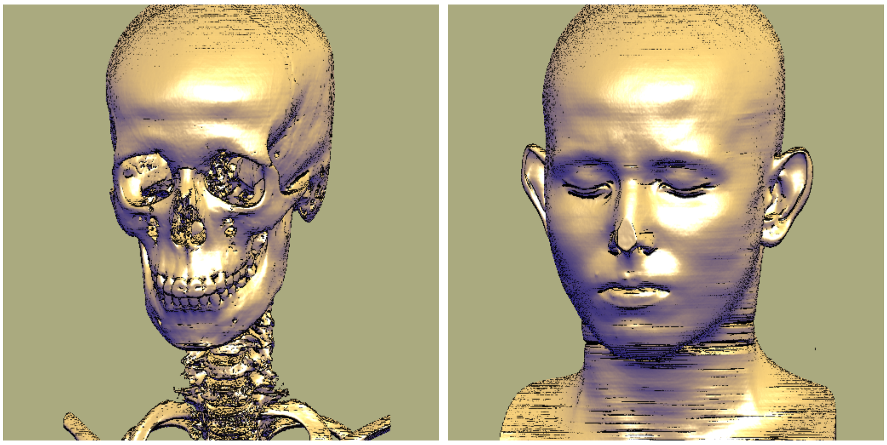


Figure 5.15: CT scan (512x512x333) with tone shading and curvature-controlled contours with ridge and valley lines specified in the (κ_1, κ_2) domain via a 2D transfer function.

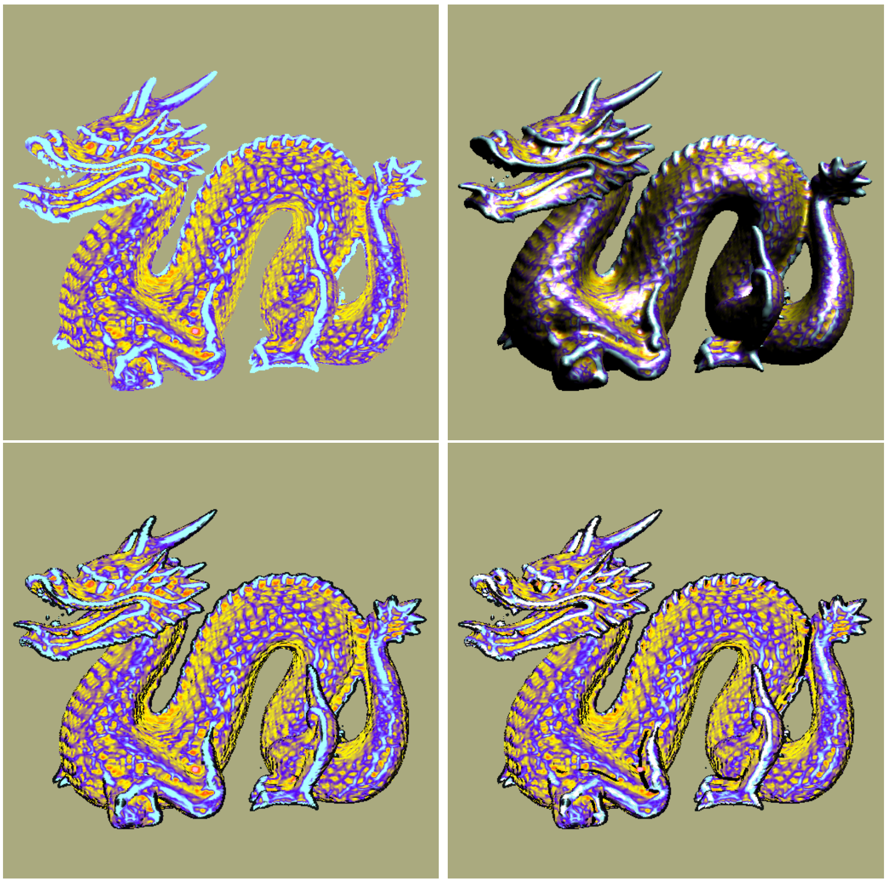


Figure 5.16: Dragon distance field (128x128x128) with colors from curvature magnitude (top, left); with Phong shading (top, right); with contours (bottom, left); with ridge and valley lines (bottom, right).

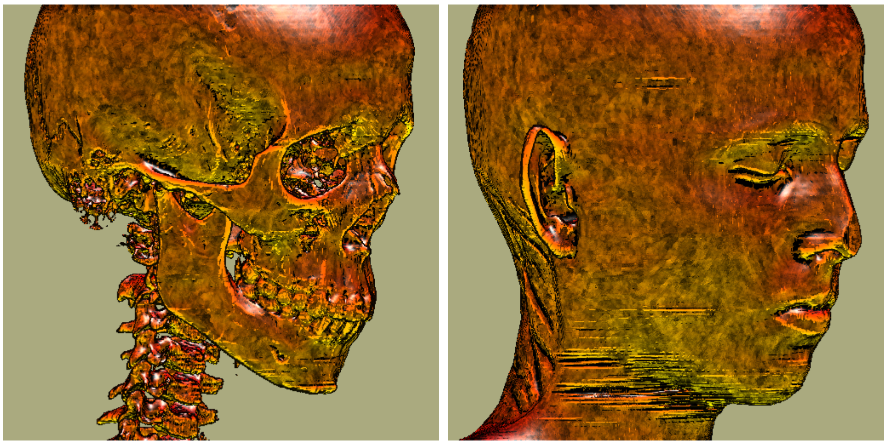


Figure 5.17: CT scan (256x256x333) with contours, ridges and valleys, tone shading, and image space flow advection to generate a noise pattern on the surface.

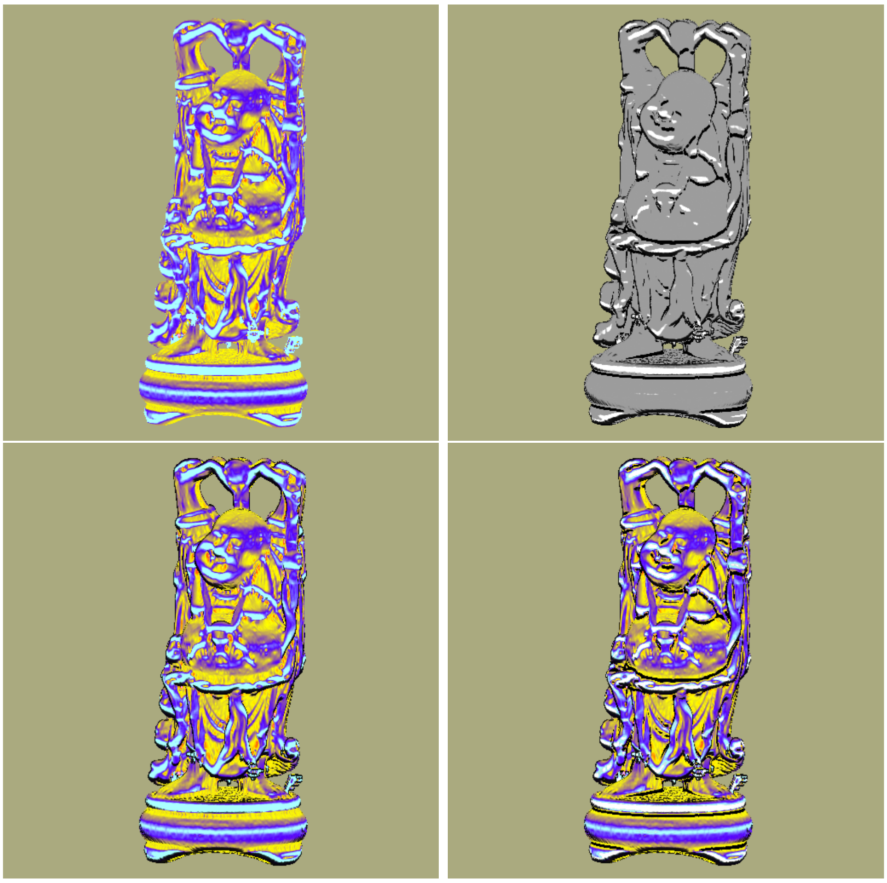


Figure 5.18: Happy Buddha distance field (128x128x128) with colors from curvature magnitude (top, left); only ridge and valley lines (top, right); with contours (bottom, left); with contours, and ridge and valley lines (bottom, right).

Chapter 6

The HVR Volume Rendering Framework

This chapter presents an overview of the major components of the HVR (*high-quality hardware volume rendering*) framework, which has been developed in the context of the research for this thesis, from an implementation-centric point of view.

The corresponding HVR renderer supports texture-based volume rendering with object-aligned slices (2D textures) with optional inter-slice tri-linear interpolation [133], view-aligned slices (3D textures) [17], and GPU-based ray casting using 3D textures without slices [78], in order to re-sample the underlying volume data. It also supports the three major approaches to classification, i.e., pre-classification, post-classification, and pre-integrated classification [24].

Slice textures can optionally be filtered with bi-cubic and tri-cubic filtering, respectively, using programmable filter kernels such as cubic B-splines and Catmull-Rom splines. High-quality filtering can be used in conjunction with all three classification modes, including pre-integration, where both slices constituting a slab are reconstructed with cubic filters.

Rendering modes include direct volume rendering with one- and two-dimensional transfer functions [70], maximum intensity projection (MIP), and non-polygonal isosurfaces [174], as well as several non-photorealistic modes including contour enhancement [16], and tone shading [28] as illustrated in chapter 5. Shading equations are evaluated on a per-fragment basis without pre-calculated lookups, yielding high-quality results. Gradients can either be pre-computed via one of three pre-process filters (central differences, Sobel filter, 4D linear regression [115]) and stored in additional volume textures, or computed on-the-fly via central differences or per-fragment convolution with derived cubic B-spline kernels.

For real-time high-quality isosurface rendering, the HVR renderer supports a full deferred shading pipeline based on tri-cubic reconstruction filters [144]. The deferred shading stages also include high-quality reconstruction of differential surface properties including second order partial derivatives and principal curvature information as described in chapter 5. High-quality curvature information can be used for a variety of surface visualization tasks and non-photorealistic rendering [67] as also illustrated in chapter 5.

A very important component of the HVR framework is the support of segmented volume data. In addition to the density volume constituting the original data set, a second volume containing object IDs can be used in order to differentiate different objects embedded in the data set. These objects can then be assigned different transfer functions, rendering modes, and even compositing modes, yielding full support for two-level volume rendering [43, 44].

6.1 Basic Structure

This section gives a high-level overview of the class structure of the HVR framework, as well as descriptions of the functionality of some of the main components, especially with respect to the rendering core.

In the following, we briefly discuss the main classes of the HVR framework. Details are discussed below. The main HVR classes are:

HVR_Interface is the main interface class, which provides access to all other classes and their functionality. It is a singleton class and handles the creation of *views* such as the 3D rendering view (**HVR_CanvasVolumeView**), and the slice view (**HVR_CanvasSliceView**).

HVR_Volume is the main encapsulation of all volumetric data including density volumes, gradient volumes, caches, and the associated OpenGL textures. It contains the embedded classes **HVR_VolumeData**, which manages the actual volume data in CPU memory; **HVR_VolumeGL**, which manages and downloads volume data as actual OpenGL textures; **HVR_VolumeCache**, which manages caching auxiliary volume data such as gradient volumes on disk; and **HVR_VolumeSetup**, which performs setup tasks such as the pre-computation of gradients.

HVR_VolumeReader manages loading volume data from disk and creating an actual **HVR_Volume** from them. Actual loading of data stored in different file formats is encapsulated in corresponding loader classes such as **HVR_VolumeReaderDAT** (for `*.dat` files) and **HVR_VolumeReaderHVR** (for `*.hvr` files) that are derived from **HVR_VolumeReader**.

HVR_View is the main interface between the GUI and the 3D volume rendering view. It separates the HVR framework from the actual GUI in used with a defined interface, and allows to exchange the GUI without changing any of the framework code. The state of the interface to the GUI is mirrored in order to allow the GUI and the actual renderer to run in separate execution threads. The GUI itself accesses variables stored in the **HVR_View** class, whereas the renderer always accesses a copy of these variables stored in the **HVR_RenderParamsView** class. In addition to per-view parameters (such as lighting parameters, viewing transformation, etc.), the **HVR_RenderParamsObject** class encapsulates data that must be stored for each object contained in a segmented volume such as the object-local transfer function and material colors.

HVR_ContextManager is the main class for managing rendering buffers and separate rendering contexts. It allows the majority of the codebase to be entirely independent from the number and type of rendering buffers used for any given rendering mode. Specifically, it allows code to be used for both standard rendering of volumes without segmentation information, and two-level volume rendering of segmented data.

HVR_ObjectManager is the main class for managing the objects contained in segmented volume data. It allows to enable and disable individual objects, and offers an interface to the actual renderer that allows it to render segmented data.

HVR_RenderTwoLevel contains rendering code specific to two-level volume rendering of segmented data. It contains the main interface and supporting code. Actual rendering code, such as fragment shaders, is contained in classes on a lower level of abstraction.

HVR_RenderRayCast contains a GPU-based ray caster that supports direct volume rendering, maximum intensity projection, and first-hit ray casting for rendering isosurfaces.

HVR_RenderImplicits contains a separate high-quality isosurface renderer that builds on the concept of deferred shading and tri-cubic filtering for all function reconstruction tasks. It is able to compute high-quality implicit curvature information on-the-fly on a per-fragment basis in deferred shading passes.

HVR_TransferFunction encapsulates the concept of a transfer function for both one-dimensional and two-dimensional transfer functions. The actual color tables are contained in the **HVR_TransferFunctionPalette** class, and the **HVR_TransferFunctionDefinition** class describes a 1D transfer function as connected piecewise linear segments, from which a color table can be generated on-demand. 2D transfer functions are described as a collection of geometric 2D primitives such as boxes and triangles with color and opacity interpolation. The transfer function definition is the representation that can be edited by the user in the *transfer function editor.*

Below, we describe the major components of the HVR framework as they are used during rendering. The order of description corresponds to the order in which these components are executed, i.e., all setup tasks are performed before actual re-sampling, shading, and compositing is invoked by rendering proxy geometry or casting rays, respectively.

Volume data representation (HVR_VolumeData)

The underlying volume data are stored in memory in a suitable format, usually already prepared for download to the graphics hardware as textures. Depending on the kind of proxy geometry used, the volume can either be stored in a single block, when view-aligned slices or ray casting together with a single 3D texture are used, or split up into three stacks of 2D slices, when object-aligned slices together with multiple 2D textures are used. In host memory, the volume is always stored in a single 3D array, which can be downloaded as a single 3D texture, or used to extract data for 2D textures on-the-fly as needed.

Depending on the complexity of the rendering mode, classification, and illumination, there may be several volumes containing all the information needed, e.g., an additional gradient volume. Likewise, the actual storage format of voxels depends on the rendering mode and the type of volume, e.g., whether the volume stores densities, gradients, gradient magnitudes, and so on. Conceptually different volumes may also be combined into the same actual volume, if possible. For example, combining gradient and density data in RGBA voxels [174]. A very important special case is the *object ID volume*, which is needed for rendering segmented data. It contains 8-bit object IDs per voxel that denote the object that the voxel belongs to.

Most data is generated only when loading data from disk. However, some data might have to be re-generated when the rendering mode or the type of gradient filter is changed. For example, the HVR framework supports three different types of pre-computed gradients: central differences, a 3^3 Sobel filter, and 4D linear regression [115]. If the type of gradient filter is changed, the gradient volume needs to be re-computed accordingly.

Volume textures (HVR_VolumeGL)

In order for the graphics hardware to be able to access all the required volume information, the volume data must be downloaded and stored in textures. At this stage, a translation from data format to OpenGL texture format might take place, if the two are not identical.

How and what textures containing the actual volume data have to be downloaded to the graphics hardware depends on a number of factors, most of all the rendering mode and type of classification, and whether 2D or 3D textures are used.

For pre-classification, the internal volume texture format consists of 8-bit color indexes, and for this reason a color lookup table must also be downloaded subsequently (**HVR_TransferFunctionPalette**). When pre-classification is not used, an intensity volume texture is downloaded instead.

In the case of view-aligned slices, only a single 3D texture per volume type (density, gradient, etc.) is downloaded. For object-aligned slicing, three different stacks of 2D textures are maintained for the three principal viewing directions.

Transfer function tables (HVR_TransferFunctionPalette)

Transfer functions are represented by one-dimensional or two-dimensional color lookup tables. How and what transfer function tables have to be downloaded to the graphics hardware depends on the type of classification that is used. They are downloaded to the hardware in basically one of two formats:

In the case of pre-classification, transfer functions are downloaded as texture palettes for on-the-fly expansion of palette indexes to RGBA colors. If post-classification is used, transfer functions are downloaded as 1D or 2D RGBA textures, depending on the dimensionality of the respective transfer function.

If pre-integration is used, the basic 1D transfer function is only used to calculate a 2D pre-integration table, but not downloaded to the hardware itself. Then, this pre-integration table is downloaded as a 2D texture instead of the original 1D transfer function.

Fragment shader configuration

Before the volume can be rendered using a specific rendering mode, the fragment shader has to be configured accordingly. The issue of how textures are stored and what they contain is crucial for the fragment shader. Likewise, the format of the shaded fragment has to correspond to what is expected by the alpha blending stage. The code that determines the operation of the fragment shader is highly dependent on the actual hardware architecture used. See section 6.3 for details on the shader usage of the HVR framework.

Blending mode configuration

The blending mode determines how a fragment is combined with the corresponding pixel in the frame buffer. In addition to the configuration of *alpha blending*, the blending mode must also account for the use of *alpha testing* if it is needed for discarding fragments that do not correspond to the desired isosurface when rendering non-polygonal isosurfaces. Although the alpha test and alpha blending are the last two steps that are actually executed by the graphics hardware in our volume rendering pipeline, they have to be configured before actually rendering any geometry.

For **direct volume rendering**, the blending mode is standard alpha blending. Since color values are usually pre-multiplied by the corresponding opacity (also known as *opacity-weighted* [179], or *associated* [6] colors), the factor for multiplication with the source color is one.

For **non-polygonal isosurfaces**, alpha testing has to be configured for selection of fragments corresponding to the desired iso-values. The comparison operator for comparing a fragment's density value with the reference value is usually `GL_GREATER`, or `GL_LESS`, since using `GL_EQUAL` is not well suited to producing a smooth surface appearance (not many interpolated density values are exactly equal to a given reference value). Alpha blending must be disabled for this rendering mode.

For **maximum intensity projection**, an alpha blending equation of `GL_MAX_EXT` must be supported, which is either a part of the imaging subset, or the separate `GL_EXT_blend_minmax` extension. On consumer graphics hardware, querying for the latter extension is the best way to determine availability of the maximum operator.

Texture unit configuration

The use of texture units corresponds to the inputs required by the fragment shader. Before rendering any geometry, the corresponding textures have to be bound. When 3D textures are used, the entire configuration of texture units usually stays the same for an entire frame. In the case of 2D textures, the textures that are bound change for each slice.

Proxy geometry rendering

The last component of the execution sequence of basic components outlined in this section is getting the graphics hardware to render geometry. This is what actually causes the generation of fragments to be shaded and blended into the frame buffer, after re-sampling the volume data accordingly.

In standard texture-based volume rendering, which is also the main approach of the HVR framework, fragments are generated by rendering *proxy geometry*. As an alternative, direct ray casting can also be used instead of rendering proxy geometry, see below.

Explicit texture coordinates are usually only specified when rendering 2D texture-mapped, object-aligned slices. In the case of view-aligned slices, texture coordinates can easily be generated automatically, by exploiting OpenGL's texture coordinate generation mechanism, which has to be configured before the actual geometry is rendered.

Vertex coordinates are specified in object-space, and transformed to view-space using the modelview matrix. If multi-texturing is used, a simple vertex program can be exploited for generating the texture coordinates for the additional units, instead of downloading the same texture coordinates to multiple units. On the latest architectures (ATI Radeon 8500+ and NVIDIA GeForce FX) it is also possible to use the texture coordinates from unit zero for texture fetch operations at any of the other units, which solves the problem of duplicate texture coordinates in a very simple way, without requiring a vertex shader or wasting bandwidth.

Ray casting instead of proxy geometry

As an alternative to re-sampling and rendering a volume by rendering proxy geometry, direct ray casting has recently also become possible on GPUs [78, 137]. The HVR framework

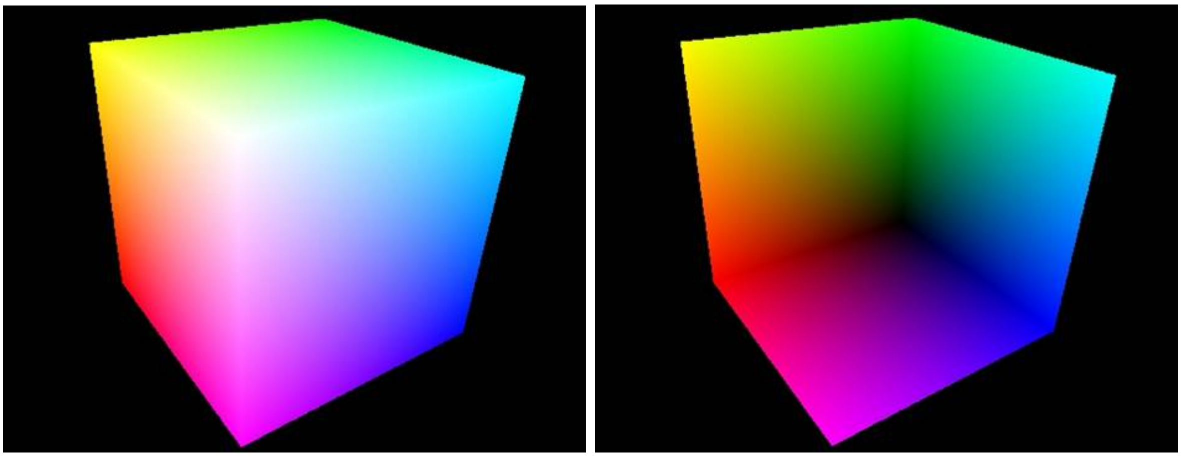


Figure 6.1: Front and back faces of the volume bounding box with 3D volume space coordinates interpolated in the RGB color channels, which are the basis for obtaining direction vectors for GPU-based ray casting.

supports shaded and unshaded direct volume rendering, first-hit ray casting, and maximum intensity projection via ray casting performed in the fragment shader.

The basic additional input for this GPU ray caster are two images of the front and back faces, respectively, of the volume bounding box, with the corresponding 3D volume space coordinates interpolated in the RGB color channels, see figure 6.1. By subtracting these two images, per-pixel, i.e., per-ray, direction vectors are obtained, which are then used to advance a front of re-sampling positions for all the corresponding rays in multiple rendering passes.

In the fragment shader, this image of ray direction vectors is used as a texture, and the current position on the ray corresponding to a given pixel is computed. The volume can then be re-sampled at that location via a simple 3D texture fetch operation.

6.2 Hardware Independence

One of the major issues in a complex rendering framework supporting a lot of different rendering modes and combinations of modes is how to keep dependence on the actual graphics hardware functionality as minimal as possible. This section first describes the functionality that is supported by the HVR framework on different hardware architectures, and then illustrates how the majority of the codebase is kept independent from the associated implementation differences.

The HVR framework supports volume rendering on the following major consumer graphics architectures, and thus needs to take their differences into account accordingly:

- **NVIDIA GeForce 256 and GeForce 2.** Basic functionality is available via the `NV_register_combiners` extension with multi-texturing of two simultaneous textures. This feature set enables the HVR rendering modes with object-aligned 2D-textured slices, also including interpolation between two adjacent slices [133]. The only supported mode for classification is pre-classification via paletted textures, since no dependent texturing operations are available on these hardware architectures.

- **NVIDIA GeForce 3 and GeForce 4.** The two major advances allowed by these architectures are the availability of 3D texturing, and the support of dependent texturing operations via the `NV_texture_shader` extension. That is, on these architectures view-aligned 3D-textured slices (using the `NV_texture_shader2` extension) can be employed in addition to object-aligned slices. Dependent texturing allows to use post-classification in addition to pre-classification, and also allows pre-integration [24]. The latter shaders need more parameters in the register combiners than previously possible and thus also use the `NV_register_combiners2` extension. The `NV_vertex_program` extension is used to handle generation of volume texture coordinates.

- **ATI Radeon 8500.** This architecture allows more flexible dependent texturing and easier handling of volume texture coordinates using the `ATI_fragment_shader` extension. Optionally, the `ATI_vertex_shader` extension is also supported in order to unify texture coordinate handling with the `NV_vertex_program` extension. Unfortunately, adding support for this graphics card to the HVR framework required all low-level shading functionality written for the `NV_register_combiners` and `NV_texture_shader` extensions to be implemented again using the `ATI_fragment_shader` extension. One drawback of this architecture is the missing support for paletted textures, and thus pre-classification is not supported.

- **ATI Radeon 9500+.** The major breakthrough towards unifying the different fragment shaders came with the support for the `ARB_fragment_program` extension on this architecture. Furthermore, this extension is required for rendering segmented volume data, which is not possible on earlier hardware. An important feature for rendering performance is the support of the early z-test on this architecture. Pre-classification is not supported on this graphics card, however, due to missing paletted texture functionality.

- **NVIDIA GeForce FX.** Apart from test shaders using the `NV_fragment_program` extension of this architecture, the HVR framework uses the same `ARB_fragment_program` code as on the ATI Radeon 9500+. Major drawbacks are the significantly lower fragment shading performance, and the fact that the early z-test does not improve performance on this architecture when rendering segmented data.

In the future, the latest NVIDIA GeForce 6 FX and ATI Radeon X800 architectures should improve performance, and further unify the fragment shader functionality, with the exception of data-dependent branching, which is only supported by the former type of hardware.

Modular design

In order to be able to support the wide range of graphics hardware and corresponding volume rendering functionality described above, the HVR framework utilizes a modular design in order to make the codebase as independent as possible from actual hardware details.

There are two distinct types of hardware-independence that are achieved by this modular structure. First, rendering modes, or specific parameters of rendering modes, are enabled or disabled at start-up depending on the detected OpenGL extensions, i.e., the feature set of the graphics hardware on which the application is running. Second, hardware-dependent code, e.g., the actual fragment shading code, and the corresponding setup must be adapted at run time to the specific API (OpenGL extension) supported by the graphics hardware on which the application is running.

The first goal is achieved by detecting the supported hardware feature set at start-up and tracking which rendering features can be supported, and which have to be disabled. This tracking/querying functionality is implemented by the **HVR_RenderModeInfo** class.

The second goal is achieved by an abstract interface layer between the actual fragment shading code, and the code that uses it, e.g., renders slice geometry. Section 6.3 contains more details about shader encapsulation in the HVR framework.

Abstract re-sampling

A major aspect of achieving hardware-independence in the HVR framework is the complete separation of *re-sampling* and *shading*.

Re-sampling is invoked by explicitly rendering proxy geometry, and hardware fragment shading is invoked automatically for all fragments resulting from rasterization. The geometry rendering code is written in standard OpenGL, and simply depends on the type of proxy geometry used, i.e., whether object-aligned or view-aligned slices are rendered. On the other hand, all fragment shading code uses various OpenGL extensions, but is independent from the way in which these fragments are generated.

Older fragment programming models (`NV_register_combiners`, `ATI_fragment_shader`) do not encapsulate knowledge of the dimension of textures that are sampled inside the shader. Thus, the corresponding shaders can be used identically for both 2D texture mapping with object-aligned slices, and 3D texture mapping with view-aligned slices. However, the current fragment programming model (`ARB_fragment_program`), includes texture dimension in the shader code itself. In this case, the HVR framework generates the corresponding actual shader code at run time via string parameterization and concatenation (see below). This makes using the same shader code for both 2D textures and 3D textures almost as trivial as in the older models.

That is, irrespective of the actual shader model (OpenGL extension) used, the corresponding shader code is only written once and can be used automatically for all different types of proxy geometry. Moreover, all other rendering parameters such as lighting and material properties are also entirely independent from proxy geometry rendering, and thus a core part of the renderer. They are also largely independent from the shader model itself, see below.

6.3 Shaders

The HVR shader model includes two major components to achieve independence from the actual graphics hardware for a large part of the codebase, and reduce the number of different fragment shaders that need to be written.

The first component is comprised of an abstract interface layer between general rendering code (geometry rendering, shader parameter specification, etc.) and the actual low-level shader code. In this way, the actual shader model need only be known at the lowest possible level of abstraction, and makes most of the code hardware-independent.

The second component creates actual shaders from *shader templates* according to parameters specified at run time, which allows to parameterize shaders even though current shader models do not include support for conditional execution of code or conditional compilation at run time.

Shader parameterization

A major problem of current shader models (in the context of this thesis: `ARB_fragment_program` and `ARB_vertex_program`) is that they are very powerful, but shaders cannot exclude code at run time via conditional execution, and the basic interfaces also do not allow conditional compilation analogously to the template mechanism available in C++, for example. This restriction leads to a large number of shaders that only differ in some parts, but could easily share most of the code. In volume rendering, this problem is especially severe, since a large number of rendering modes often differ only in small details and the general number of fragment shaders is very high.

In the HVR framework, for example, a fragment shader for direct volume rendering has to work for object-aligned slicing (using 2D textures) and view-aligned slicing (using 3D textures). It has to deal with unshaded and shaded volume rendering. Gradients can be pre-computed and stored in textures, or they can be computed on-the-fly. When rendering segmented data, fragments might need to be rejected depending on the corresponding object ID sampled from an ID texture. This ID texture might be filtered in the fragment shader, or it might be used without filtering for performance reasons. Moreover, shader code might have to be adapted to the resource limits of the actual hardware. Even though the `ARB_fragment_program` API is supported by both NVIDIA and ATI architectures, the available resources such as the maximum number of instructions are very different. This list could easily be continued. Basically, all of these possibilities can be built into a single shader, because often a large percentage of the shader code is identical. However, this is not directly possible, which can lead to a lot of code duplication, bug fixing, and a generally much higher amount of required code maintenance.

The HVR framework avoids these problems to a large extent by using a simple way of *run time shader parameterization*. Shader code is written and added to a shader on a line-by-line (instruction-by-instruction) basis in a *shader template* written in C++. Whenever a shader is needed at run time, the C++ code generating the string of the entire shader code decides for each line whether it should be added to the actual shader or not depending on parameters supplied by the user of the shader. In essence, this amounts to a very simple system of conditional string concatenation, which builds the actual shader code string that will be compiled by the OpenGL driver entirely at run time. Shader strings are only generated on-demand and exactly once, by tracking whether the actual code string has already been generated and compiled previously.

6.4 User Interface

This section gives an overview of the major user interface components of the default GUI of the HVR framework. However, exchanging the GUI for an alternative one can be done without touching any code in the HVR framework itself. The GUI attaches to the HVR framework via the **HVR_Interface** and **HVR_View** classes.

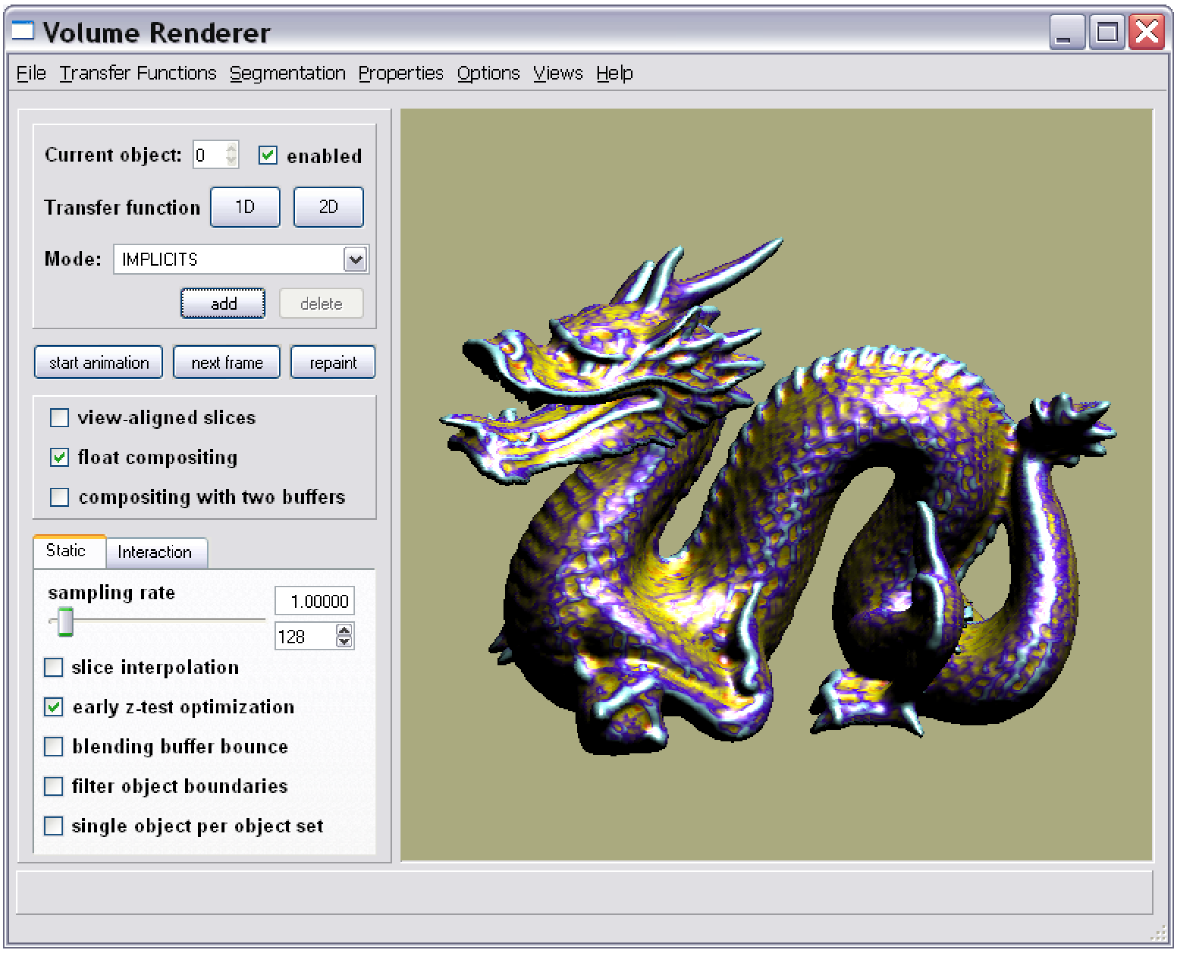
The authoritative state of all user interface elements is maintained by the **HVR_View** class. The GUI itself only displays this state, and requests changes. These changes might also be denied, e.g., when a value is set out of range, which might depend on the actual graphics hardware in use and thus can cannot be determined by the GUI itself. This separation also ensures that the GUI does not contain actual functionality, and can thus be exchanged easily while leaving all existing functionality in place.

Likewise, the renderer itself requests only a copy of the entire user interface state from the **HVR_View** class at the beginning of each frame, and uses a copy of this state during rendering. This allows for thread-safety and separate GUI and rendering threads.

Volume view

Figure 6.2 shows the main user interface and 3D volume view of the default HVR GUI. The major components accessible from this view are the enabling and disabling of segmented objects, and corresponding per-object properties such as the transfer function and the rendering mode. The HVR framework also supports animated volume data (by loading multiple volume frames specified in a single text file), on-the-fly switching between object-aligned and view-aligned slicing, floating point compositing, and correct two-level compositing with two rendering buffers, or approximate compositing with only a single buffer.

Many properties are available twice, once for a *static* mode, and once for an *interaction* mode. The former is used for high-quality settings, and the latter for lower-quality settings that are used during user interactions such as rotating the volume. The sampling rate can be selected directly (which is kept consistent for all kinds of proxy geometry as well as ray



Current object: 0 ☑ enabled

Mode: IMPLICITS

☐ view-aligned slices

☑ float compositing

☐ compositing with two buffers

sampling rate 1.00000

128

☐ slice interpolation

☑ early z-test optimization

☐ blending buffer bounce

☐ filter object boundaries

☐ single object per object set

Figure 6.2: The HVR GUI main view containing the 3D volume view.

casting), or via the number of slices (which might change depending on the view when object-aligned slices and rectangular volume dimensions are used).

Transfer function editor

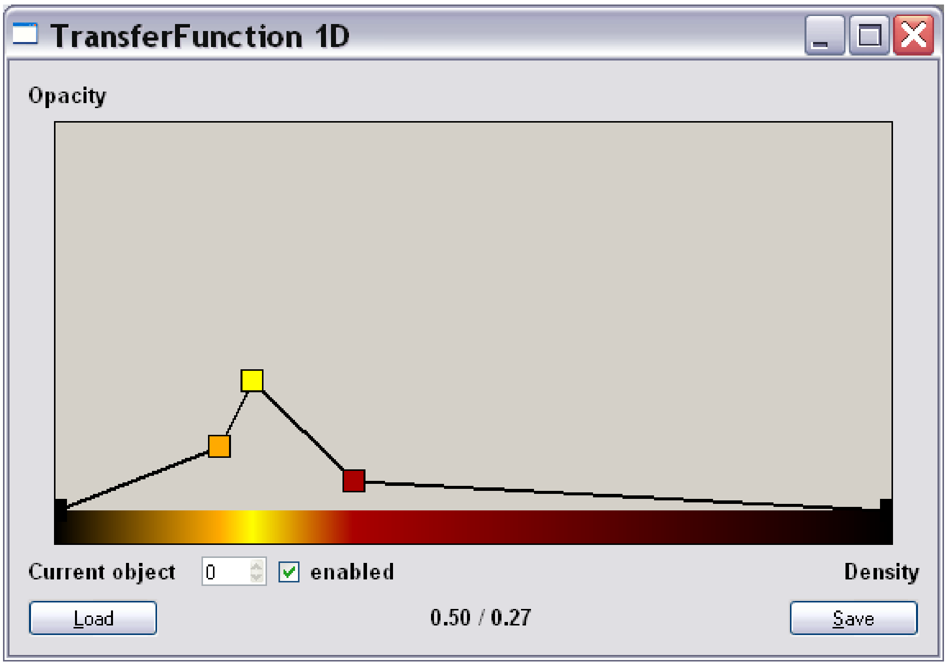
Figure 6.3 shows the 1D transfer function editor. The transfer function is modified by dragging nodes with the mouse. These nodes are assigned a color (clicking opens a color chooser), and an opacity (their position on the vertical axis), and connected by linear segments. This piecewise linear specification is converted to a transfer function table every time it is modified. The actual transfer function table is shown at the bottom of this window.

The transfer function editor also allows to switch between different segmented objects for easy specification of per-object transfer functions, as well as loading and saving transfer functions individually.

Properties and Options

Figures 6.4 and 6.5 show the properties and options dialogs, respectively. The properties dialog contains:

- The *lighting* tab for specifying background color, ambient, diffuse, and specular light colors, intensities, and reflection coefficients.
- The *clipping* tab for specification of six per-object clipping planes.
- The *NPR parameters* tab (shown in figure 6.4) allows specification of parameters for tone shading, contour rendering, and maximum intensity projection (MIP).



Current object 0 ☑ enabled

0.50 / 0.27

Figure 6.3: The HVR GUI 1D transfer function editor.

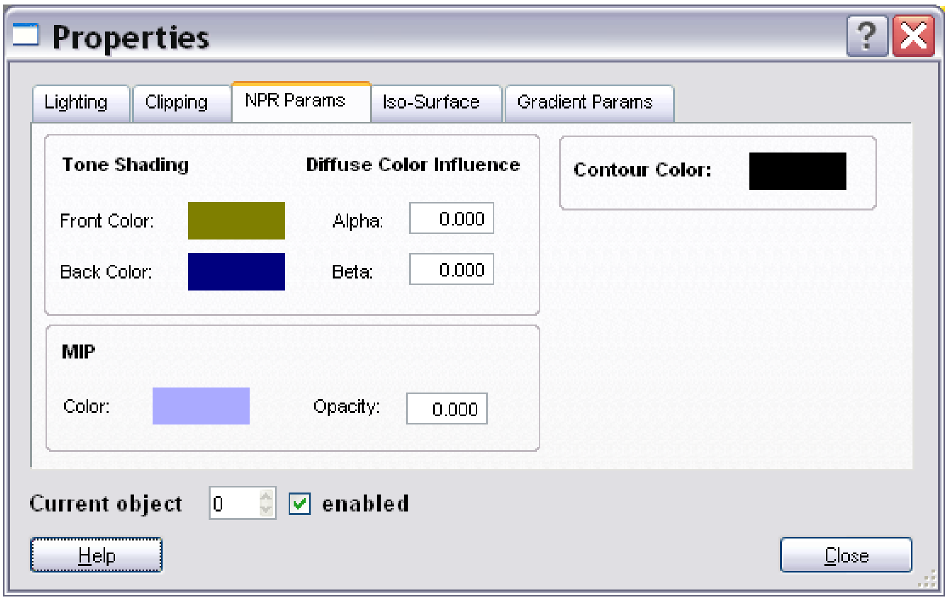
- The *isosurface* tab allows to set iso-values for multiple isosurfaces, as well as their front- and back-facing colors.
- The *gradient params* tab allows specification of different gradient filters separately for the gradient direction and the gradient magnitude. It also allows to iteratively smooth gradients that have been computed or loaded.

The options dialog contains:

- The *general* tab for global settings such as the global compositing mode for two-level rendering, and flags such as whether to display the frame rate or volume bounding box.
- The *objects* tab for handling segmented objects.
- The *filtering* tab for choosing the kind of reconstruction filter for different tasks.
- The *curvature* tab (shown in figure 6.5) for settings regarding the high-quality curvature visualization of isosurfaces.
- The *animation* tab for animated volumes, which allows to select individual frames, change the replay rate, etc.
- The *data* tab, which determines data pre-processing such as automatic down-sampling when loading a volume, conversion of volume formats (e.g., from 16-bit voxels to 8-bit voxels), and caching converted data on disk.
- The *stereo* tab for rendering a stereoscopic images of the volume, separated in even and odd screen rows or columns, for stereo display.

Additional functionality

In addition to the functionality exported in the GUI, the 3D volume view supports a large number of key assignments, especially for testing and debugging, as well as functionality that is used infrequently.



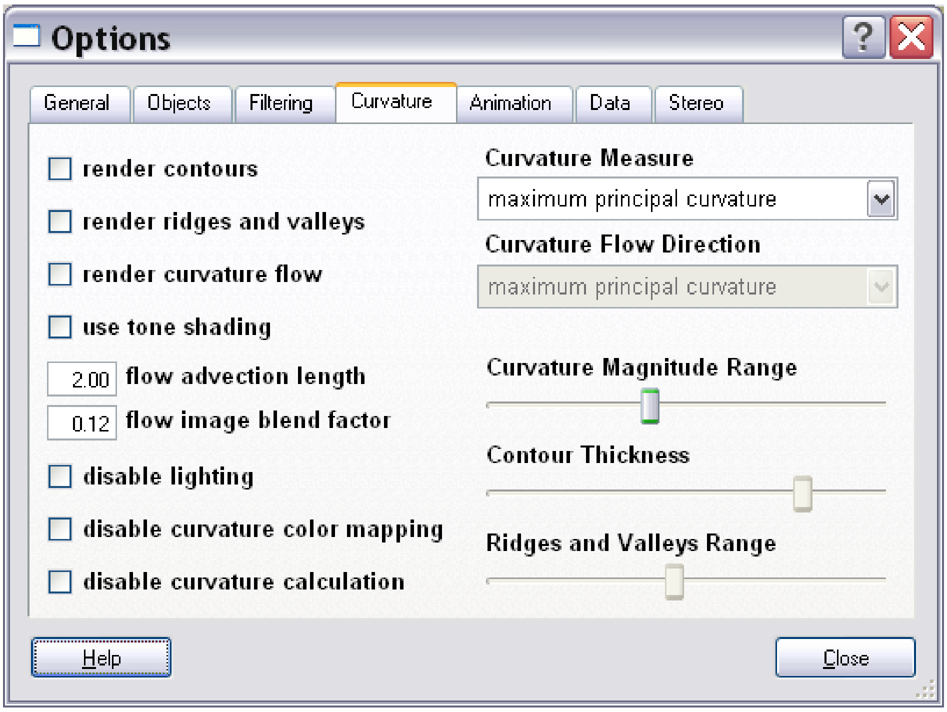
Alpha: 0.000

Beta: 0.000

Opacity: 0.000

Current object 0 ☑ enabled

Figure 6.4: The HVR GUI properties dialog showing the *NPR parameters* tab.



☐ render contours

☐ render ridges and valleys

☐ render curvature flow

☐ use tone shading

2.00 flow advection length

0.12 flow image blend factor

☐ disable lighting

☐ disable curvature color mapping

☐ disable curvature calculation

Curvature Measure: maximum principal curvature

Curvature Flow Direction: maximum principal curvature

Figure 6.5: The HVR GUI options dialog showing the *curvature* tab.

Chapter 7

Summary

The tremendous evolution of consumer graphics hardware, since the addition of highly programmable shading units often referred to as GPUs (graphics processing units), has created a platform for real-time high-quality rendering methods that were previously only possible in off-line rendering.

We introduce several algorithms for leveraging the parallel processing power of consumer graphics hardware, ranging from the early NVIDIA GeForce 256 graphics cards to the current state-of-the-art NVIDIA GeForce FX and ATI Radeon 9500+ architectures, for high-quality visualization and filtering of texture maps and segmented volumetric data, and demonstrate high-quality rendering and shading including non-photorealistic effects.

7.1 High-Quality Filtering

A very important basic step of volume rendering is the reconstruction of the original continuous signal from sampled data. In the general case, this reconstruction step is usually accomplished by filtering the original data via convolution with a reconstruction filter.

In texture-based volume rendering on consumer graphics hardware, the volume data are stored in one or multiple texture maps. We show how these textures can be filtered with arbitrary convolution filters, which allows using higher-order filter kernels in order to attain high reconstruction quality. The presented method can also be used to filter all kinds of texture maps in a variety of applications.

Fundamentally, we evaluate the general filter convolution sum, which can be stated for the one-dimensional case as:

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} f_i \cdot h(x - i) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} f_i \cdot w_i(x) \tag{7.1}$$

where $g(x)$ is the output at re-sampling position x, f_i is the discrete input texture, $h(x)$ is the continuous filter kernel, $m = n/2$ is half the filter width when n is the order (cubic: $n = 4$), and the $w_i(x)$ are the n weights corresponding to x.

In the simplest and most general case, we evaluate equation 7.1 in multiple rendering passes using two simultaneous textures each. The first texture contains the unmodified input texture, and the second texture one part of the filter kernel that we call a *filter tile.* Filter tiles split up the filter kernel into regions of unit extent, and hence the number of tiles is equal to the size of the filter. Likewise, the number of rendering passes without optimizations is equal to the number of filter tiles and thus the size of the filter kernel.

In contrast to filters for image processing, where the number of values representing the kernel is equal to its size, we sample each filter tile with a resolution of 64 or 128 samples per dimension. Thus, this representation of filter kernels has much higher sampling resolution than image processing filters and we also call them *tiled high-resolution filters*.

The basic algorithm can be modified to take various properties of the filter kernel into account, as well as exploiting features of the target hardware architecture. The most important of these filter properties are symmetry and separability. Graphics hardware with a large number of simultaneous textures in a single rendering pass is able to evaluate complex convolution sums in a single rendering pass, e.g., cubic convolution on an ATI Radeon 9800. On current GPUs, the simplest and fastest approach for cubic convolution is to store an entire cubic filter kernel in the four channels of a single 1D RGBA texture, and retrieve four filter weights at once with a single texture fetch from this kernel texture. See figure 7.2.

Although the basic method is intended for *magnification filters*, i.e., the case where a single texel maps to multiple pixels, it can be extended to work with MIP-mapped textures, which allows to also use higher-order reconstruction filters in the case of *texture minification*. We do this by introducing the concept of a *meta MIP-map* that stores all the information necessary for performing a per-pixel adaptation of the filter kernel in screen space.

Stated in OpenGL terminology, our filtering framework allows to substitute both of the hardware-native texture reconstruction filters, the magnification and the minification filter, by a higher-order filter. This enables, for example, implementation of a `GL_CUBIC` filter for the OpenGL magnification filter specification, and `GL_CUBIC_MIPMAP_NEAREST` and `GL_CUBIC_MIPMAP_LINEAR` filters for the minification filter specification.

See figure 7.1 for a quality comparison of linear interpolation and cubic filtering, in both the cases of magnification and minification.

The basic building block of texture-based volume rendering is rendering individual slices through the volume. On these slices, the volume data are reconstructed just as in standard texture mapping, which allows to substitute the hardware-native interpolation with a higher-

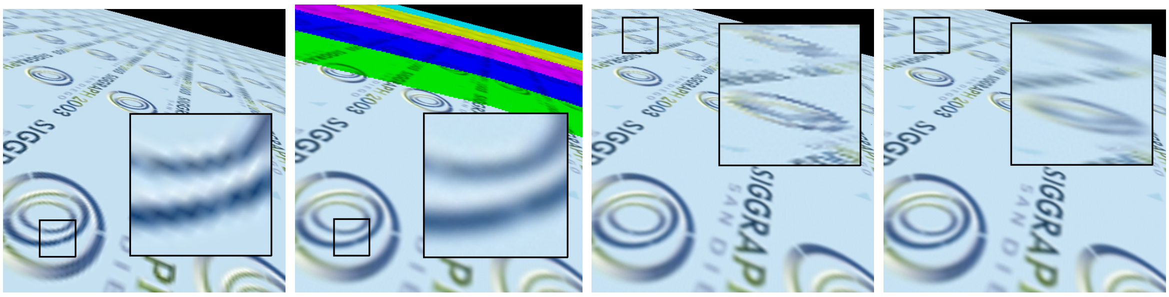


Figure 7.1: Higher-order vs. linear filtering of textures in both the cases of texture magnification and minification, respectively. (left) texture magnification using hardware-native linear interpolation; (second left) texture magnification with a cubic B-spline filter using our framework; MIP-map levels are color-coded in this image; (second right) in lower-resolution MIP-map levels, the cubic filter degenerates to nearest-neighbor interpolation if it is not adapted to the level of texture minification; (right) cubic filtering in all MIP-map levels, including magnification (base level of MIP-map) and minification (other levels).

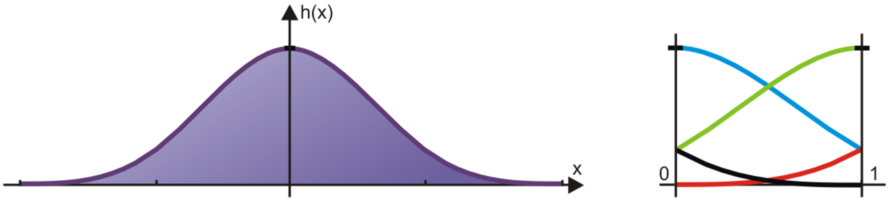


Figure 7.2: A 1D cubic filter kernel (shown here: cubic B-spline) is sampled and stored in the four channels of a single RGBA texture map. A single texture fetch operation retrieves the four weights needed for the evaluation of the filter convolution sum at any re-sampling location.

order reconstruction filter.

In the case of *object-aligned slices*, which use standard 2D textures, 2D convolution with a higher-order filter kernel can be used in order to improve reconstruction quality. For *view-aligned slices*, which require the volume data to be stored in a 3D texture map, 3D convolution with a higher-order filter can be used analogously.

In order to generate volume renderings from individual texture-mapped slices, these slices have to be composited in either back-to-front or front-to-back order.

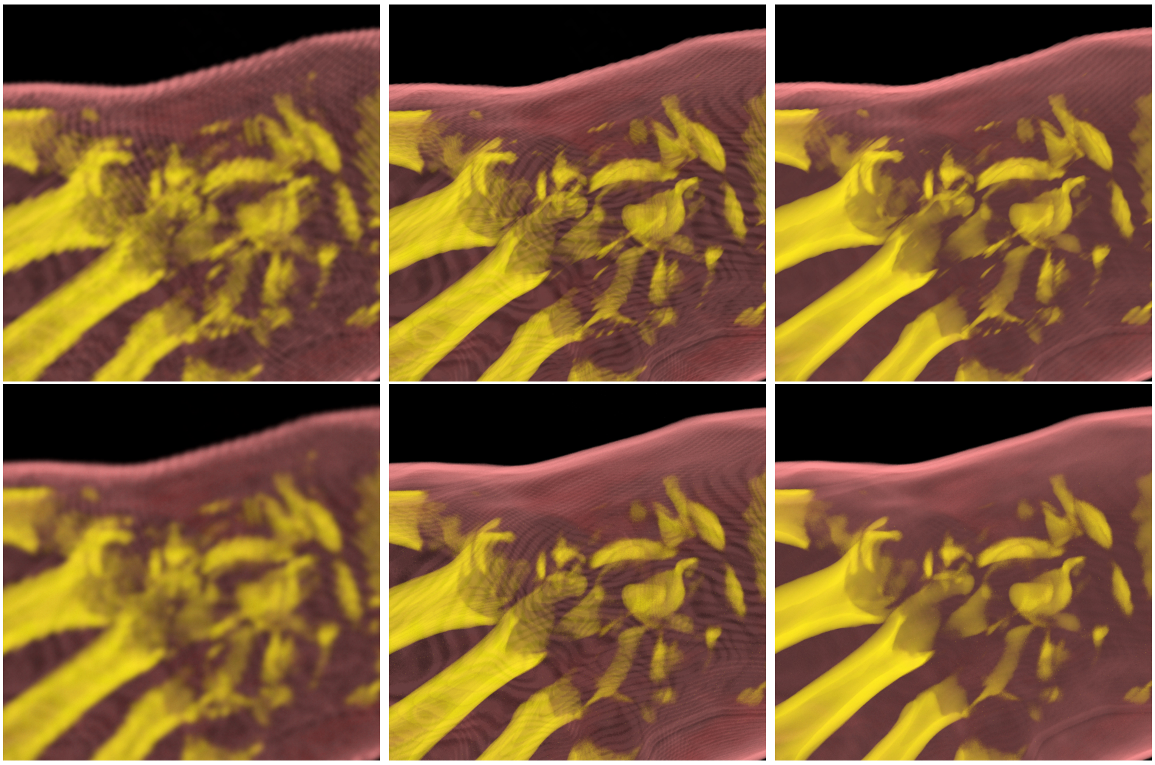


Figure 7.3: Hand (256x128x256) volume rendering with bi-linear (top row) vs. bi-cubic (bottom row) reconstruction of object-aligned slices. Left column with pre-classification; center column with post-classification; right column with pre-integration.

Figure 7.3 compares bi-linear and bi-cubic slice interpolation for each of the major three classification modes: pre-classification, post-classification, and pre-integrated classification [24]. Pre-classification maps densities to colors and opacities before interpolation takes place, which leads to strong blurring artifacts (figure 7.3, first column), especially when the transfer function contains high frequencies. Post-classification interpolates densities and then maps the interpolation results to colors and opacities, which yields much sharper images (figure 7.3, center column). This preserves high frequencies in the transfer function much better. Pre-integrated classification decouples the frequencies in the transfer function entirely from the frequencies in the density volume, which yields even sharper images and is generally considered to be the best approach for low sampling rates (figure 7.3, right column).

The pre-integrated renderings also illustrate that even when pre-integration is used, cubic filtering can still improve the final result significantly. The two approaches of pre-integration and higher-order slice filtering could be considered to improve quality in an orthogonal manner. Better filtering improves reconstruction quality within the individual slices, whereas pre-integration improves reconstruction quality with respect to the transfer function between slices, i.e., in the direction orthogonal to the slices themselves.

7.2 Volume Rendering of Segmented Data

One of the most important goals in volume rendering, especially when dealing with medical data, is to be able to visually separate and selectively enable specific objects of interest contained in a single volumetric data set. A very powerful approach to facilitate the perception of individual objects is to create explicit object membership information via *segmentation* [159]. The process of segmentation determines a set of voxels that belong to a given object of interest, usually in the form of one or several *segmentation masks*.

An especially powerful approach is to combine different non-photorealistic and traditional volume rendering methods in a single volume rendering. When segmentation information is available, different objects can be rendered with individual per-object rendering modes, which

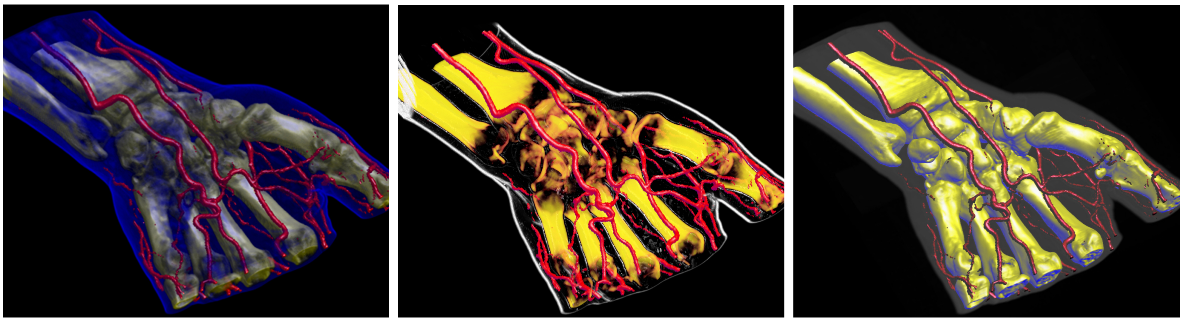


Figure 7.4: Segmented hand data set (256x128x256) with three objects: skin, blood vessels, and bone. Two-level volume rendering integrates different transfer functions, rendering and compositing modes: (left) all objects rendered with shaded DVR; the skin partially obscures the bone; (center) skin rendered with non-photorealistic contour rendering and MIP compositing, bones rendered with DVR, vessels with tone shading; (right) skin rendered with MIP, bones with tone shading, and vessels with shaded isosurfacing; the skin merely provides context.

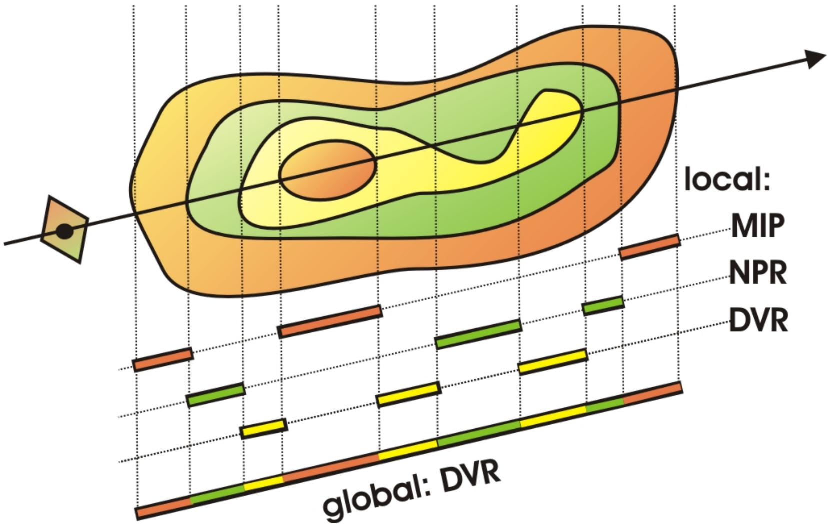


Figure 7.5: A single ray corresponding to a given image pixel is allowed to pierce objects that use their own object-local compositing mode. The contributions of different objects along a ray are combined with a single global compositing mode. Rendering a segmented data set with these two conceptual levels of compositing (local and global) is known as *two-level volume rendering*.

allows to use specific modes for structures they are well suited for, as well as separating *focus* from *context*. See figure 7.4 for examples.

We demonstrate how segmented volumetric data sets can be rendered efficiently and with high quality on current consumer graphics hardware. The segmentation information for object distinction can be used at multiple levels of sophistication, and we integrate all of these different possibilities into a single coherent hardware volume rendering framework.

First, different objects can be rendered with the same rendering technique (e.g., DVR), but with different transfer functions. Separate per-object transfer functions can be applied in a single rendering pass even when object boundaries are filtered during rendering.

Second, different objects can be rendered using different hardware fragment shaders. This allows easy integration of methods as diverse as non-photorealistic and direct volume rendering, for instance. Although each distinct fragment shader requires a separate rendering pass, multiple objects using the same fragment shader with different rendering parameters can effectively be combined into a single pass. When multiple passes cannot be avoided, the cost of individual passes is reduced drastically by executing expensive fragment shaders only for those fragments active in a given pass. These two properties allow highly interactive rendering of segmented data sets, since even for data sets with many objects usually only a couple of different rendering modes are employed.

Finally, different objects can also be rendered with different compositing modes, e.g., alpha blending and maximum intensity projection (MIP), for their contribution to a given pixel. These per-object compositing modes are object-local and can be specified independently for each object. The individual contributions of different objects to a single pixel can be combined via a separate global compositing mode. This two-level approach to object compositing [44] has proven to be very useful in order to improve perception of individual objects. See figure 7.5.

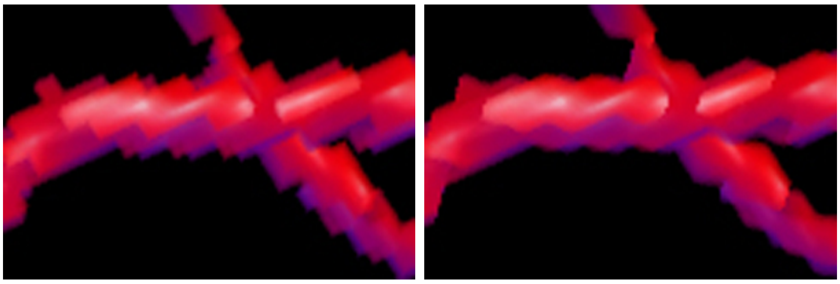


Figure 7.6: Object boundaries with voxel resolution (left) vs. object boundaries determined per-fragment with linear filtering (right).

The major novel contributions of our method are:

- A systematic approach to minimizing both the number of rendering passes and the performance cost of individual passes when rendering segmented volume data with high quality on current GPUs. Both filtering of object boundaries and the use of different rendering parameters such as transfer functions do not prevent using a single rendering pass for multiple objects. Even so, each pass avoids execution of the corresponding potentially expensive fragment shader for irrelevant fragments by exploiting the early z-test. This reduces the performance impact of the number of rendering passes drastically.
- An efficient method for mapping a single object ID volume to and from a domain where filtering produces correct results even when three or more objects are present in the volume. The method is based on simple 1D texture lookups and able to map and filter blocks of four objects simultaneously. See figure 7.6 for a comparison of using an object ID volume with and without filtering, respectively.
- An efficient object-order algorithm based on simple depth and stencil buffer operations that achieves correct compositing of objects with different per-object compositing modes and an additional global compositing mode. The result is conceptually identical to being able to switch compositing modes for any given group of samples along the ray for any given pixel.

7.3 Non-Photorealistic Volume Rendering

Non-photorealistic rendering techniques, e.g., rendering styles imitating artistic illustration, have established themselves as a very powerful tool for conveying a specific meaning in rendered images, especially in renderings of surfaces. In recent years, the interest in adapting existing NPR techniques to volumes and creating new entirely volumetric NPR models has increased significantly.

We demonstrate high-quality tone shading [28] of volumes taking transfer function information into account, and per-pixel evaluation of a simple volumetric contour rendering model [16].

We further illustrate a real-time high-quality rendering pipeline for rendering isosurfaces with *deferred shading* and deferred computation of differential implicit surface properties,

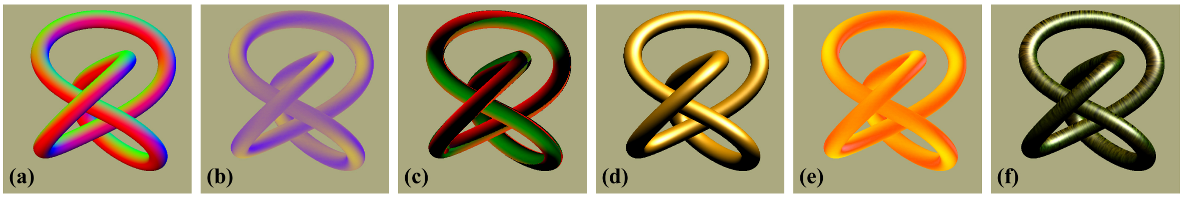


Figure 7.7: Example image space rendering passes of deferred isosurface shading. Surface properties such as (a) the gradient, (b) principal curvature magnitudes (here: κ_1), and (c) principal curvature directions can be reconstructed. These properties can be used in shading passes, e.g., (d) Blinn-Phong shading, (e) color coding of curvature measures (here: $\sqrt{\kappa_1^2+\kappa_2^2}$), and (f) advection of flow along principal curvature directions.

such as first and second partial derivatives, including the gradient for shading computations, and principal curvature magnitudes and directions.

When deferred shading is employed, the actual data specified in object space, i.e., in our case the volume, are first processed without any potentially expensive shading computations in order to determine actually visible pixels. Shading and other computations are then performed in subsequent rendering passes in image space for these pixels only.

Computing implicit surface curvature is a powerful tool for isosurface investigation and non-photorealistic rendering of isosurfaces. Tri-cubic filtering is the minimum requirement for computation of high-quality principal curvature estimates, and we demonstrate the applicability of the resulting curvature information in a variety of non-photorealistic isosurface rendering modes.

Principal curvature magnitudes can be visualized on an isosurface by mapping them to colors via one-dimensional or two-dimensional transfer function lookup textures.

One-dimensional curvature transfer functions. Simple color mappings of first or second principal curvature magnitude via 1D transfer function lookup tables can easily be computed during shading. The same approach can be used to depict additional curvature measures directly derived from the principal magnitudes, such as mean curvature $(\kappa_1+\kappa_2)/2$ or Gaussian curvature $\kappa_1\kappa_2$. See figure 7.8 for an example.

Two-dimensional curvature transfer functions. Transfer functions in the 2D domain of both principal curvature magnitudes (κ_1, κ_2) are especially powerful, since color specification in this domain allows to highlight different structures on the surface [50], including ridge and valley lines [57, 67]. Curvature magnitude information can also be used to implement silhouette outlining with constant screen space thickness [67]. See figures 7.8, 7.9, and 7.10 for examples.

Direct mappings of principle curvature directions to RGB colors are hard to interpret, see figure 7.7(c), for example.

However, principal curvature directions on an isosurface can be visualized using image-based flow visualization [162]. In particular, flow can be advected on the surface entirely in image space [82]. These methods can easily be used in real-time, complementing the capability to generate high-quality curvature information on-the-fly, which also yields the underlying, potentially unsteady, "flow" field in real-time. See figure 7.7(f). In this case, it is natural to perform per-pixel advection guided by the floating point image containing principal direction vectors instead of warping mesh vertex or texture coordinates.

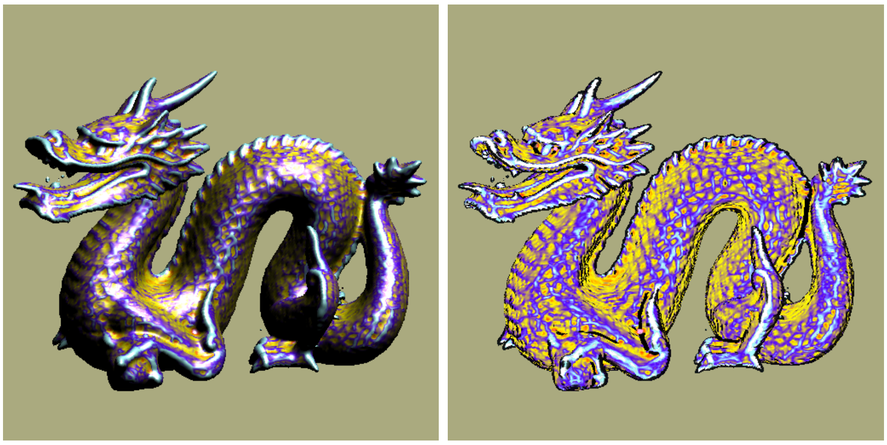


Figure 7.8: Dragon distance field (128x128x128) with colors from curvature magnitude and Blinn-Phong shading (left); and with contours, and ridge and valley lines (right).

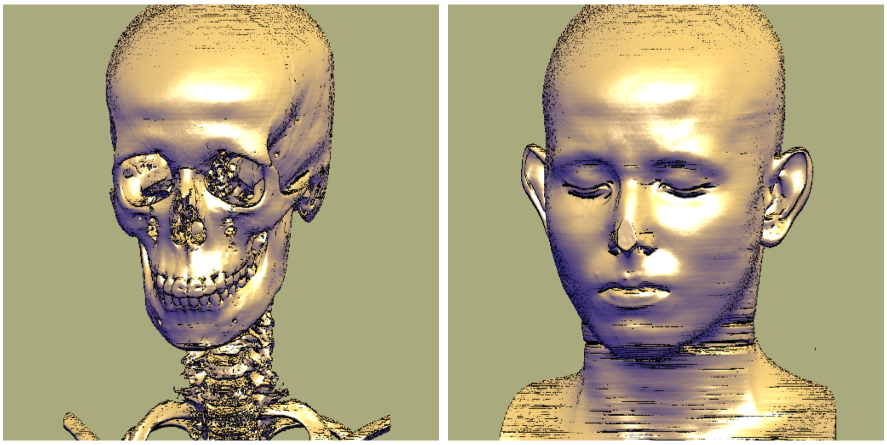


Figure 7.9: CT scan (512x512x333) with tone shading and curvature-controlled contours with ridge and valley lines specified in the (κ_1, κ_2) domain via a 2D transfer function.

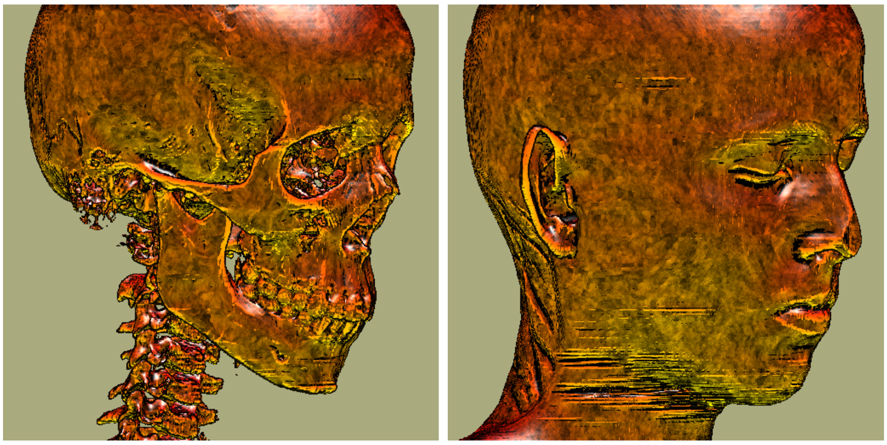


Figure 7.10: CT scan (256x256x333) with contours, ridges and valleys, tone shading, and image space flow advection to generate a noise pattern on the surface.

Chapter 8

Conclusions

The techniques presented in this thesis allow real-time rendering of textures and volume data with an image quality that is usually only possible in off-line rendering.

The presented framework for high-quality filtering of textures has a lot of applications due to its basic nature. Practically all applications involving texture mapping with 1D, 2D, and 3D textures can make use of higher-order convolution filters instead of the linear interpolation supported natively by the hardware.

The extension of the basic approach for magnification filters toward minification filters with MIP-mapping is an important step toward establishing higher-order filters as an attractive alternative in a wide variety of applications.

Although the basic method allows to evaluate completely arbitrary convolution filters during texture mapping, choosing cubic filters as a substitute for linear interpolation provides a very good trade-off between performance and quality. Cubic filtering can also be done in a single rendering pass on the latest graphics hardware generation, which greatly simplifies the use of these filters in volume rendering and general texture mapping with transparency, since it obviates the need for filtering in intermediate buffers before blending.

Moreover, cubic filters are very well suited to be used for reconstruction of differential properties such as the first partial derivatives (the gradient), and second partial derivatives (the Hessian) in a volume, or on an isosurface of a volume, respectively.

A very interesting application of these reconstructed properties is real-time computation of implicit surface curvature, for which cubic filters are the minimum requirement in order to achieve adequate quality.

An important observation and concept with respect to potentially expensive shading computations, including the derivation of differential properties, is the notion of *deferred shading.* On the latest generation of graphics hardware, computations that were always thought to be far away from real-time performance can actually be computed in real-time if they are only computed for visible or actually contributing pixels. Naturally, this is especially true for shading isosurfaces, where for each ray into the volume only a single shading computation must be performed.

Implicit surface curvature proves to be very useful for a variety of non-photorealistic effects. NPR techniques are well suited to rendering surfaces or isosurfaces, but can also be adapted effectively to displaying entire volumes.

Non-photorealistic volume rendering is especially powerful in conjunction with segmented volume data, where NPR and traditional volume rendering methods can be combined on the basis of individual objects. In this way, it provides a powerful additional cue to separating focus, i.e., regions of interest, from context, which is only provided as auxiliary visual information.

When rendering segmented data, the inherent inflexibility of graphics hardware is a major problem. Traditionally, even fragments that will be culled before modifying the frame buffer are computed and shaded in their entirety. This degrades performance for rendering different objects with different rendering modes greatly, since these modes have to use their own respective fragment shaders. If shading equations are then evaluated for all fragments, even those belonging to objects that have been assigned another shading equation, a lot of computational effort is wasted on fragments that will be culled anyway.

However, this inflexibility can be circumvented on the latest generation of GPUs. In order to avoid the performance problem of shading culled fragments, early depth (or z) testing can be employed, which discards fragments that will be culled later on in the pipeline before the shading computation is started.

In addition to performance considerations, filtering the boundaries of segmented objects can be crucial for image quality, depending on the transfer function or iso-value used. Usually, object boundaries are simply used with nearest-neighbor interpolation. The latest generation of GPUs, however, is flexible enough to allow linear filtering of object boundaries without necessarily increasing the number of rendering passes.

In summary, we conclude that the programmability and flexibility of consumer graphics hardware (GPUs) has reached a level where it starts to rival what is possible in traditional CPU-based rendering even with respect to flexibility.

Considering the fact that performance is often an enabling factor, the use of GPUs with carefully tuned algorithms can enable a degree of flexibility that is not possible on CPUs due to non-interactive performance.

A very good example is the use of cubic filtering, which is only made practical for interactive volume rendering by the texture mapping and filtering power of GPUs. Although cubic filters can in principle be used in CPU volume rendering, their use is restricted to off-line rendering, and is never considered for interactive systems.

We can conclude that the computational power of GPUs allows to bridge the gap between traditional high-quality, but off-line, rendering, and interactive rendering, enabling not only higher performance but also higher quality in real-time rendering systems.

Appendix A

Meta MIP-Map Details

This appendix elaborates on practical implementation details of the meta MIP-map concept introduced in section 3.2, especially with respect to memory usage of the meta MIP-map, and dependencies on details of graphics hardware architectures.

Reducing meta MIP-map storage

At first, the memory overhead needed for using the meta MIP-map concept seems considerable. In practice, however, it can be moderate to extremely low, depending on the features supported by the graphics hardware.

Two major factors contribute to the memory overhead of using meta MIP-maps. First, we will concentrate on the number of different meta MIP-maps needed. Second, we will consider the size and format of values stored in the meta MIP-map that determine the size of a single map.

Meta MIP-map dimensions and size

We observe that in order to guarantee that correct values are fetched from the meta MIP-map, in principle it must have dimensions and size identical to the actual image texture. However, a single meta MIP-map can always be used directly for all possible image textures of the same dimension and size without any additional considerations, since it contains only information about MIP-map geometry and is not related to the actual contents of a given image texture MIP-map at all.

Simulating different sizes

The first way to reduce the number of meta MIP-maps in an application using lots of different texture sizes is to use the `GL_TEXTURE_BASE_LEVEL` parameter. This value essentially strips higher-resolution MIP-map levels and treats the corresponding texture as though it would contain only lower resolutions. This makes it possible to have a single meta MIP-map of sufficient resolution for all image textures of a given aspect ratio. For using the meta MIP-map together with an image texture of a given resolution, the `GL_TEXTURE_BASE_LEVEL` simply has to be set to match the resolution of the image texture. Using this optimization, there only need to be as many meta MIP-maps as there are different aspect ratios used by the application.

Simulating different aspect ratios and dimensions

The second way to reduce the number of meta MIP-maps entirely obviates the need for meta MIP-maps of different aspect ratios, but requires full access to the Jacobian matrix in the pixel shader (see below). This essentially amounts to simulating all possible aspect ratios and dimensions with a single 1D meta MIP-map. The only requirement is that the 1D meta MIP-map has as many levels as the biggest texture used by the application. If this is satisfied, a single meta MIP-map suffices for all possible image textures.

In order to simulate a meta MIP-map aspect ratio that is different from the actual aspect ratio, it is necessary to have access to the Jacobian matrix in the pixel shader. The Jacobian contains the partial derivatives of texture coordinates (s and t in the 2D case) with respect to screen space coordinates x and y [47, 181, 182]:

$$\begin{pmatrix} \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} \end{pmatrix}$$

In order to simulate a different aspect ratio, the corresponding coordinate (s or t) has to multiplied by its actual size.

For example, if we are using a meta MIP-map of width 256 and height 1 and want to use it together with an image texture of width 256 and height 128, we have to multiply both $\frac{\partial t}{\partial x}$ and $\frac{\partial t}{\partial y}$ by 128. The original partials have been calculated with respect to a height of 1, and by simple multiplication by the simulated height of 128 we get the same behavior as a real 256 x 128 texture would exhibit. Vice versa, if we want to use this meta MIP-map of width 256 and height 1 with an image texture of width 128 and height 256, we have to multiply the partials $\frac{\partial s}{\partial x}$ and $\frac{\partial s}{\partial y}$ by 0.5, and the partials $\frac{\partial t}{\partial x}$ and $\frac{\partial t}{\partial y}$ by 256. All other possible aspect ratios can be simulated analogously.

The base level of the meta MIP-map must be set in order to match the size of the longest dimension of the image texture, i.e., in both examples above `GL_TEXTURE_BASE_LEVEL` would be 0. If the meta MIP-map would be of width 512, however, the base level would have to be 1 in both examples.

On the GeForce FX, the `NV_fragment_program` extension allows full access to the Jacobian via the `DDX` (yielding the left column vector of the Jacobian) and `DDY` (yielding the right column vector of the Jacobian) instructions. After correcting the aspect ratio as outlined above, the modified Jacobian is used for accessing the meta MIP-map using the `TXD` instruction of `NV_fragment_program`, which accepts explicitly specified partial derivatives in order to determine the MIP-map level to use.

Note that although we do not know the actual MIP-map level that will be used for a given pixel when modifying the Jacobian, we know how to modify it in order to simulate arbitrary aspect ratios of 2D textures with a single 1D texture. The same concept could be used for MIP-mapping 3D textures using only a single 1D meta MIP-map.

Meta MIP-map texture format

Naturally, the number of channels and number format used by the meta MIP-map also influences its memory overhead. The meta information can either be stored directly in the meta MIP-map itself, or it can contain indexes into an additional 1D texture that stores the actual information.

In the latter case, the meta MIP-map contains only a single channel with 8-bit entries (e.g., `GL_INTENSITY8`), which minimizes the memory overhead due to the map's texture format. It introduces a slight overhead into the pixel shader, however, since an additional 1D texture access with computed texture coordinates becomes necessary.

Note that even if the aspect ratio simulation outlined above cannot be used on a given hardware architecture, using a single-channel 8-bit texture format instead of floating point textures with multiple channels reduces the memory overhead considerably.

Hardware architecture idiosyncracies

Both of the major contemporary consumer graphics hardware architectures, the ATI Radeon 9500+ and the NVIDIA GeForce FX, have major differences in their feature set as it relates to using meta MIP-maps.

Floating point textures

First, if the meta MIP-map should contain floating point data, it must be possible to use MIP-mapped floating point textures. This is possible on the ATI architecture, where all OpenGL texture formats are supported with floating point counterparts, but not on the NVIDIA architecture. The GeForce FX supports floating point data only for rectangular textures, which are not allowed to contain MIP-maps.

Therefore, a meta MIP-map on the GeForce FX is required to store indexes into an additional look-up texture, which in this case has to be a rectangular texture of height 1. Also, the inter-level interpolation weight cannot be retrieved at full precision.

Partial derivatives

Second, the simulation of different aspect ratios can only be used on the GeForce FX, since it supports access to the Jacobian matrix through the `NV_fragment_program` extension. The ATI Radeon 9500+ support only the `ARB_fragment_program` extension, which currently does not allow access to the Jacobian and only contains the possibility to bias the LOD level used for a given texture fetch instead of being able to specify the full partial derivatives.

Appendix B

OpenGL Extensions

This appendix briefly summarizes basic OpenGL extensions that are useful for hardware-accelerated volume rendering.

`GL_EXT_blend_minmax`

This extension augments the OpenGL alpha blending capabilities by minimum (`GL_MIN_EXT`) and maximum (`GL_MAX_EXT`) operators, which can be activated via the `glBlendEquationEXT()` function. When one of these special alpha blending modes is used, a fragment is combined with the previous contents of the frame buffer by taking the minimum, or the maximum value, respectively.

For volume rendering, this capability is needed for maximum intensity projection (MIP), where pixels are set to the maximum density along a "ray."

`GL_EXT_paletted_texture`, `GL_EXT_shared_texture_palette`

In hardware-accelerated volume rendering, the volume itself is usually stored in texture maps with only a single channel. Basically, there are two OpenGL texture formats that are used for these textures, both of which consume one byte per voxel.

First, a volume can be stored in intensity textures (`GL_LUMINANCE` as external, `GL_INTENSITY8` as internal format). In this case, each voxel contains the original density values, which are subsequently mapped to RGBA values by post-classification, or pre-integration, for example.

Second, a volume can be used for rendering with pre-classification. In this case, storing the volume in an RGBA texture (`GL_RGBA` as external, `GL_RGBA8` as internal format) would be possible. However, this consumes four times the texture memory that is actually necessary, since the mapping from density to RGBA can easily be performed by the hardware itself. In order to make this possible, paletted textures need to be supported via `GL_EXT_paletted_texture`. Using this extension, a texture is stored as 8-bit indexes into a color palette (`GL_COLOR_INDEX` as external, `GL_COLOR_INDEX8` as internal format). The palette itself consists of 256 entries of four bytes per entry (for RGBA).

The `GL_EXT_paletted_texture` extension by itself needs a single palette for each individual texture, which must be downloaded via a `glColorTableEXT()` function call. However, in volume rendering with 2D slices, all slice textures actually use the same palette.

In order to share a single palette among multiple textures and download it only once, the `GL_EXT_shared_texture_palette` extension can be used. Using this extension, only a single palette need be downloaded with a `glColorTableEXT()` function call in conjunction with the `GL_SHARED_TEXTURE_PALETTE_EXT` parameter.

`GL_EXT_texture_env_dot3`

Although more flexible and powerful functionality is exposed by the NVIDIA register combiners (`GL_NV_register_combiners`) and various fragment shader extensions (`GL_ARB_fragment_program`, `GL_ATI_fragment_shader`, ...), it is not always desired to incur the development overhead of specifying a full register combiner setup or fragment shader, when only a simple per-fragment dot-product is needed. This extension extends the modes that can be used in the standard texture environment for combining the incoming color with the texture color by a simple three-component dot-product.

Acknowledgments

First I would like to thank my advisors Helwig Hauser and Prof. Eduard Gröller (*The Master!*, probably of all and everything :), whose input and patient supervision provided a very important foundation for this work. Helwig's relentless efforts were especially important for making this happen. A very special thank-you also goes to Prof. Thomas Ertl from the University of Stuttgart for accepting a position on my PhD committee.

I am highly indebted to my main collaborator Thomas Theußl, and my former students Christoph Berger and Ivan Viola. The contribution of Christoph is especially important to me, and his memory will always be with me. My current students Henning Scharsach and Caroline Langer are also a very important part of a stimulating environment.

I would also like to thank Prof. Werner Purgathofer (*The President!* of the VRVis) for fostering a very relaxed but highly professional atmosphere at the Institute of Computer Graphics and Algorithms, and establishing the VRVis Research Center.

I would also like to extend a very warm *Thank You!* to my friends and many of my current and former colleagues at the VRVis Research Center and the Institute of Computer Graphics and Algorithms; in alphabetical order: Jiří Bittner, Katja Bühler, Helmut Doleisch, Jiří Hladůvka, Konrad Karner, Andreas Klaus, Robert Kosara, Alexandra La Cruz, Bob Laramee, André Neubauer, Matej Mlejnek, Lukas Mroz, Berk Özer, Harald Piringer, Georg Rothwangl, Georg Stonawski, Anna Vilanova í Bartroli, Michael Wimmer, Christina Winkler, and Christopher Zach.

I am also indebted to my international friends, colleagues, and collaborators; in alphabetical order: Klaus Engel, Prof. Markus Gross, Gordon Kindlmann, Joe Kniss, Aaron Lefohn, Prof. Torsten Möller, Christof Rezk-Salama, Christian Sigg, Daniel Weiskopf, and Prof. Rüdiger Westermann. Especially the help of Christof and Christian has contributed significantly to the quality of this thesis.

Most of this work has been done in the VRVis Research Center, which is funded in part by the Kplus program of the Austrian government. The medical data sets are courtesy of Tiani Medgraph. The distance field data sets have been generated by Christian Sigg. NVIDIA provided a GeForce 3 graphics card, and Michael Doggett of ATI provided a Radeon 9700 graphics card.

Curriculum Vitae

Dipl.-Ing. Markus Hadwiger,
born on 18 November 1974, in Vienna, Austria.

Education

- PhD student at the Vienna University of Technology since November 2000.
- In October 2000, I received my Dipl.-Ing. degree from the Vienna University of Technology, where I studied Computer Science from October 1994 to October 2000. The diploma thesis, *Design and Architecture of a Portable and Extensible Multiplayer 3D Game Engine*, describes the design and implementation of an entire game engine.
- In June 1994, I graduated from a technical college in Vienna 1010, Austria, with focus on electronics and hardware ("Elektronik und Nachrichtentechnik"). The final project resulted in the hardware for a programmable six-channel sound card for the PC (ISA) architecture, including firmware and PC software.
- From 1985-1989, I attended secondary school in Vienna 1100, Austria.
- From 1981-1985, I attended primary school in Vienna 1100, Austria.

Jobs

- Junior researcher in the *Effective Medical Visualization* group at the VRVis Research Center for Virtual Reality and Visualization, since January 2004.
- Junior researcher in the *Basic Research on Visualization* group at the VRVis Research Center for Virtual Reality and Visualization, from September 2000 to December 2003.
- Teaching assistant at the Institute of Computer Graphics and Algorithms, Vienna University of Technology, for the lectures *Computer Graphics 2 and 3* (summer semesters 1997, 1998, 1999, and 2000), *Visualization* (winter semester 1999), and *Virtual Reality* (winter semesters 1997 and 1998)
- In 1994, I worked on the soundsystem and soundtrack for the computer game *Oldtimer*, which was published by *Max Design*, Austria.
- In the summer of 1993, I worked for *Philips Communications and Processing* in Vienna, Austria.

- In 1992, I worked on graphics and other low-level assembly language code for the computer game *Elysium*, which was published by *Magic Bytes*, Germany.
- In the summer of 1991, I worked for *Siemens Austria* in Vienna, Austria.

Publications and Talks

See the Bibliography in this thesis for my publications, as well as `http://www.vrvis.at/vis/staff/hadwiger/publs.html` for an up-to-date list of publications and talks.

Professional Activities

Reviewer for

- IEEE Visualization Conference (Vis).
- Joint Eurographics/IEEE Symposium on Visualization (VisSym).
- IEEE Transactions on Visualization and Computer Graphics (TVCG).
- IEEE Computer Graphics and Applications (CG&A).
- Winter School on Computer Graphics (WSCG).
- Spring Conference on Computer Graphics (SCCG).
- Central European Seminar on Computer Graphics (CESCG).

Session chair at

- Winter School on Computer Graphics (WSCG) 2003.
- Central European Seminar on Computer Graphics (CESCG) 2002.
- Spring Conference on Computer Graphics (SCCG) 2000.

Bibliography

[1] ATI web page. http://www.ati.com/.

[2] C. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *Proceedings of IEEE Visualization '97*, pages 167–ff., 1997.

[3] A. Barr. Ray tracing deformed surfaces. In *Proceedings of SIGGRAPH '86*, pages 287–296, 1986.

[4] K. Bjorke. High-quality filtering. In *GPU Gems*, pages 391–415. Addison Wesley Professional, 2004.

[5] J. Blinn. Hyperbolic interpolation. *IEEE Computer Graphics and Applications*, 12(4):89–94, 1992.

[6] J. Blinn. Jim blinn's corner: Image compositing—theory. *IEEE Computer Graphics and Applications*, 14(5):83–87, 1994.

[7] J. Blinn. Three wrongs make a right. *IEEE Computer Graphics and Applications*, 15(6):90–93, 1995.

[8] J. Blinn. Floating-point tricks. *IEEE Computer Graphics and Applications*, 17(4), 1997.

[9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proceedings of SIGGRAPH 2003*, pages 917–924, 2003.

[10] U. Bordoloi and H.-W. Shen. Space efficient fast isosurface extraction for large datasets. In *Proceedings of IEEE Visualization 2003*, pages 201–208, 2003.

[11] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[12] Y.-J. Chiang, C. Silva, and W. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization '98*, pages 167–174, 1998.

[13] P. Cignoni, P. Marino, E. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[14] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Proceedings of IEEE Visualization '93*, pages 261–267, 1993.

[15] B. Csébfalvi and E. Gröller. Interactive volume rendering based on a bubble model. In *Proceedings of Graphics Interface 2001*, pages 209–216, 2001.

[16] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. In *Proceedings of Eurographics 2001*, pages 452–460, 2001.

[17] T. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill, 1993.

[18] F. Dachille, K. Kreeger, B. Chen, I. Bittner, and A. Kaufman. High-quality volume rendering using texture mapping hardware. In *Proceedings of Eurographics/SIGGRAPH Graphics Hardware Workshop 1998*, 1998.

[19] R. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proceedings of SIGGRAPH '88*, pages 65–74, 1988.

[20] D. Ebert, C. Morris, P. Rheingans, and T. Yoo. Designing effective transfer functions for volume rendering from photographic volumes. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):183–197, 2002.

[21] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing & Modeling: A Procedural Approach.* Third Edition, Morgan Kaufman Publishers, 2003.

[22] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, pages 195–202, 2000.

[23] K. Engel, M. Hadwiger, J. Kniss, and C. Rezk-Salama. *High-Quality Volume Graphics on Consumer PC Hardware.* Course Notes for Course #42 at SIGGRAPH 2002, ACM SIGGRAPH, 2002.

[24] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Graphics Hardware 2001*, pages 9–16, 2001.

[25] K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proceedings of IEEE Visualization '99*, pages 139–146, 1999.

[26] R. Fernando, S. Fernandez, K. Bala, and D. Greenberg. Adaptive shadow maps. In *Proceedings of SIGGRAPH 2001*, pages 387–390, 2001.

[27] R. Fernando and M. Kilgard. *The Cg Tutorial - The Definitive Guide to Programmable Real-Time Graphics.* Addison Wesley, 2003.

[28] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of SIGGRAPH '98*, pages 447–452, 1998.

[29] B. Gooch and A. Gooch. *Non-Photorealistic Rendering.* A.K. Peters Ltd., 2001.

[30] General Purpose Computations on GPUs (GPGPU) web page. http://www.gpgpu.org/.

[31] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. VOTS: VOlume doTS as a point-based representation of volumetric data. In *Proceedings of Eurographics 2004*, 2004.

[32] S. Guthe and W. Strasser. Real-time decompression and visualization of animated volume data. In *Proceedings of IEEE Visualization 2001*, pages 349–356, 2001.

[33] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization 2002*, pages 53–60, 2002.

[34] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 301–308, 2003.

[35] M. Hadwiger, H. Hauser, and T. Möller. Quality issues of hardware-accelerated high-quality filtering on PC graphics hardware. In *Proceedings of the 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) 2003*, pages 213–220, 2003.

[36] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2001*, pages 105–112, 2001.

[37] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering of solid textures. In *SIGGRAPH 2001 Conference Abstracts and Applications*, page 194, 2001.

[38] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Mip-mapping with procedural and texture-based magnification. In *SIGGRAPH 2003 Sketches and Applications*, 2003.

[39] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2002*, pages 155–162, 2002.

[40] P. Haeberli and M. Segal. Texture mapping as a fundamental drawing primitive. In *Proceedings of Fourth Eurographics Workshop on Rendering*, pages 259–266, 1993.

[41] M. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of Graphics Hardware 2002*, pages 109–118, 2002.

[42] M. Harris and A. Lastra. Real-time cloud rendering. In *Proceedings of Eurographics 2001*, pages 76–84, 2001.

[43] H. Hauser, L. Mroz, G.-I. Bischi, and E. Gröller. Two-level volume rendering - fusing MIP and DVR. In *Proceedings of IEEE Visualization 2000*, pages 211–218, 2000.

[44] H. Hauser, L. Mroz, G.-I. Bischi, and E. Gröller. Two-level volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):242–252, 2001.

[45] P. Heckbert and H. Moreton. Interpolation for polygon texture mapping and shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111, 1991.

[46] P. Heckbert. Filtering by repeated integration. In *Proceedings of SIGGRAPH '86*, pages 317–321, 1986.

[47] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping.* Master's thesis, University of California at Berkeley, 1989.

[48] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of transfer functions with stochastic search techniques. In *Proceedings of IEEE Visualization 96*, pages 227–234, 1996.

[49] K. Hillesland, S. Molinov, and R. Grzeszczuk. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. In *Proceedings of SIGGRAPH 2003*, pages 925–934, 2003.

[50] J. Hladůvka, A. König, and E. Gröller. Curvature-based transfer functions for direct volume rendering. In *Proceedings of Spring Conference on Computer Graphics 2000*, pages 58–65, 2000.

[51] J. Hladůvka, A. König, and E. Gröller. Salient representation of volume data. In *Proceedings of the Joint Eurographics/IEEE TVCG Symposium on Visualization 2001*, pages 203–211, 2001.

[52] J. Hladůvka. *Derivatives and Eigensystems for Volume-Data Analysis and Visualization.* PhD thesis, Vienna University of Technology, 2001.

[53] M. Hopf and T. Ertl. Accelerating 3D convolution using graphics hardware. In *Proceedings of IEEE Visualization '99*, pages 471–474, 1999.

[54] M. Hopf and T. Ertl. Accelerating morphological analysis with graphics hardware. In *Workshop on Vision, Modelling, and Visualization (VMV) 2000*, 2000.

[55] M. Hopf and T. Ertl. Hardware accelerated wavelet transformations. In *Proceedings of Eurographics/IEEE TCVG Symposium on Visualization 2000*, 2000.

[56] J. Huang, K. Mueller, N. Shareef, and R. Crawfis. FastSplats: Optimized splatting on rectilinear grids. In *Proceedings of IEEE Visualization 2000*, pages 219–226, 2000.

[57] V. Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Proceedings of IEEE Visualization '95*, pages 52–59, 1995.

[58] V. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *Proceedings of SIGGRAPH '97*, pages 109–116, 1997.

[59] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1996.

[60] G. James. Operations for hardware-accelerated procedural texture animation. In *Game Programming Gems 2*, pages 497–509. Charles River Media, 2001.

[61] T.J. Jankun-Kelly and K.-L. Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[62] A. Kadosh, D. Cohen-Or, and R. Yagel. Tricubic interpolation of discrete surfaces for binary volumes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):580–586, 2003.

[63] D. Kalra and A. Barr. Guaranteed ray intersections with implicit surfaces. In *Proceedings of SIGGRAPH '89*, pages 297–306, 1989.

[64] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany. The imagine stream processor. In *Proceedings of IEEE International Conference on Computer Design*, pages 282–288, 2002.

[65] R. Keys. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(6):1153–1160, 1981.

[66] G. Kindlmann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of IEEE Volume Visualization '98*, pages 79–86, 1998.

[67] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of IEEE Visualization 2003*, pages 513–520, 2003.

[68] G. Kindlmann. *Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering.* Masters thesis, Cornell University, 1999.

[69] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of IEEE Visualization 2001*, pages 255–262, 2001.

[70] J. Kniss, G. Kindlmann, and C. Hansen. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[71] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, pages 109–116, 2002.

[72] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[73] J. Kniss, S. Premoze, M. Ikits, A. Lefohn, C. Hansen, and E. Praun. Gaussian transfer functions for multi-field volume visualization. In *Proceedings of IEEE Visualization 2003*, pages 497–504, 2003.

[74] L. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of SIGGRAPH 2001*, pages 57–66, 2001.

[75] A. König and E. Gröller. Mastering transfer function specification by using volumepro technology. In *Proceedings of Spring Conference on Computer Graphics 2001*, pages 279–286, 2001.

[76] M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *Proceedings of IEEE Visualization 2001*, pages 215–222, 2001.

[77] M. Kraus and T. Ertl. Adaptive texture maps. In *Proceedings of Graphics Hardware 2002*, pages 7–15, 2002.

[78] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 287–292, 2003.

[79] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *Proceedings of SIGGRAPH 2003*, pages 908–916, 2003.

[80] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH '94*, pages 451–458, 1994.

[81] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of IEEE Visualization '99*, pages 355–361, 1999.

[82] B. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *Proceedings of IEEE Visualization 2003*, pages 131–138, 2003.

[83] B. Laramee, J. van Wijk, B. Jobard, and H. Hauser. ISA and IBFVS: Image space based visualization of flow on surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 2004.

[84] A. Lefohn, J. Cates, and R. Whitaker. Interactive, GPU-based level sets for 3D brain tumor segmentation. In *Proceedings of Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2003.

[85] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 75–82, 2003.

[86] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 2003.

[87] A. Lefohn and R. Whitaker. A GPU-based, three-dimensional level set solver with curvature flow. Technical Report UUCS-02-017, 2002, University of Utah School of Computing, 2002.

[88] J. Leven, J. Corso, J. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3D textures. In *Proceedings of IEEE Visualization 2002*, pages 37–44, 2002.

[89] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[91] B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction to Volume Rendering*. Prentice-Hall, New Jersey, 1998.

[92] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Proceedings of IEEE Visualization '98*, 1998.

[93] Y. Livnat, H.-W. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[94] W. Li and A. Kaufman. Accelerating volume rendering with texture hulls. In *Proceedings of IEEE VolVis 2002*, pages 115–122, 2002.

[95] W. Li and A. Kaufman. Texture partitioning and packing for accelerating texture-based volume rendering. In *Proceedings of Graphics Interface 2003*, pages 81–88, 2003.

[96] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 317–324, 2003.

[97] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of SIGGRAPH '87*, pages 163–169, 1987.

[98] E. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR) 2002*, 2002.

[99] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. In *Proceedings of IEEE Visualization 2002*, pages 211–218, 2002.

[100] J. Marks, B. Andalman, P. Beardsley, and H. Pfister. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of SIGGRAPH 97*, pages 389–400, 1997.

[101] W. Mark and K. Proudfoot. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of Graphics Hardware 2001*, pages 57–64, 2001.

[102] S. Marschner and R. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of IEEE Visualization '94*, pages 100–107, 1994.

[103] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[104] M. McCool. Homomorphic factorization for BRDFs. In *Proceedings of SIGGRAPH 2001*, pages 171–178, 2001.

[105] J. McCormack, R. Perry, K. Farkas, and N. Jouppi. Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH '99*, pages 243–250, 1999.

[106] T. McReynolds, D. Blythe, B. Grantham, and S. Nelson. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 2000 course notes*, 2000.

[107] M. Meißner, J. Huang, D. Bartz, K. Müller, and R. Crawfis. A practical evaluation of four popular volume rendering algorithms. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 81–90, 2000.

[108] D. Mitchell and A. Netravali. Reconstruction filters in computer graphics. In *Proceedings of SIGGRAPH '88*, pages 221–228, 1988.

[109] T. Möller, R. Machiraju, K. Müller, and R. Yagel. Classification and local error estimation of interpolation and derivative filters for volume rendering. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 71–78, 1996.

[110] T. Möller, R. Machiraju, K. Müller, and R. Yagel. Evaluation and Design of Filters Using a Taylor Series Expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.

[111] T. Möller, K. Müller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[112] H. Müller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):115–134, 1999.

[113] K. Museth, D. Breen, R. Whitaker, and A. Barr. Level set surface editing operators. In *Proceedings of SIGGRAPH 2002*, pages 330–338, 2002.

[114] Z. Nagy, J. Schneider, and R. Westermann. Interactive volume illustration. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2002*, 2002.

[115] L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4d linear regression. In *Proceedings of Eurographics 2000*, pages 351–357, 2000.

[116] NVIDIA. Cg effects browser 5.0, bicubic texture filtering example, 2002. See http://developer.nvidia.com/.

[117] NVIDIA web page. http://www.nvidia.com/.

[118] M. Olano, S. Mukherjee, and A. Dorbie. Vertex-based anisotropic texturing. In *Proceedings of Graphics Hardware 2001*, pages 95–98, 2001.

[119] A. Oppenheim and R. Schafer. *Digital Signal Processing.* Prentice Hall, 1975.

[120] A. Paeth. Proper treatment of pixels as integers. In *Graphics Gems I*, pages 249–256. Academic Press, 1990.

[121] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization '98*, pages 233–238, 1998.

[122] Parsec - there is no safe distance. http://www.parsec.org/.

[123] D. Peachey. Solid texturing of complex surfaces. In *Proceedings of SIGGRAPH '85*, pages 279–286, 1985.

[124] M. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, pages 425–432, 2000.

[125] V. Pekar, R. Wiemker, and D. Hempel. Fast detection of meaningful isosurfaces for volume data visualization. In *Proceedings of IEEE Visualization 2001*, pages 223–230, 2001.

[126] K. Perlin. An image synthesizer. In *Proceedings of SIGGRAPH '85*, pages 287–296, 1985.

[127] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, and R. Machiraju. The transfer function bake-off. In *Proceedings of IEEE Visualization 2000*, pages 523–526, 2000.

[128] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. Sobierajski Avila, K. Martin, R. Machiraju, and J. Lee. The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(3):16–22, 2001.

[129] M. Pharr. Fast filter width estimates with texture maps. In *GPU Gems*, pages 417–424. Addison Wesley Professional, 2004.

[130] T. Porter and T. Duff. Compositing digital images. In *Proceedings of SIGGRAPH '84*, pages 253–259, 1984.

[131] K. Proudfoot, W. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, pages 159–170, 2001.

[132] H. Qu, A. Kaufman, R. Shao, and A. Kumar. A framework for sample-based rendering with o-buffers. In *Proceedings of IEEE Visualization 2003*, pages 441–448, 2003.

[133] 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 2000*, pages 109–118, 2000.

[134] C. Rezk-Salama, P. Hastreiter, J. Scherer, and G. Greiner. Automatic adjustment of transfer functions for 3D volume visualization. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2000*, pages 357–364, 2000.

[135] P. Rheingans and D. Ebert. Volume illustration: Nonphotorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, pages 253–264, 2000.

[136] R. Rost. *OpenGL Shading Language.* Addison Wesley, 2004.

[137] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of VisSym 2003*, pages 231–238, 2003.

[138] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of IEEE Visualization 2000*, pages 109–116, 2000.

[139] Y. Sato, C.-F. Westin, A. Bhalerao, S. Nakajima, N. Shiraga, S. Tamura, and R. Kikinis. Tissue classification based on 3D local intensity structures for volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):160–180, 2000.

[140] A. Schilling, G. Knittel, and W. Strasser. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, May 1996.

[141] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 293–300, 2003.

[142] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. http://www.opengl.org.

[143] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249–252, 1992.

[144] C. Sigg, M. Hadwiger, M. Gross, and K. Bühler. Real-time high-quality rendering of isosurfaces. Technical Report TR-VRVis-2004-015, VRVis Research Center, 2004.

[145] C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 83–90, 2003.

[146] T. Strothotte and S. Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation.* Morgan Kaufmann, 2002.

[147] R. Strzodka. Virtual 16 bit precise operations on RGBA8 textures. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2002*, pages 171–178, 2002.

[148] A. Sud, M. Otaduy, and D. Manocha. DiFi: Fast 3D distance field computation using graphics hardware. In *Proceedings of Eurographics 2004*, 2004.

[149] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher. Geometric surface smoothing via anisotropic diffusion of normals. In *Proceedings of IEEE Visualization 2002*, pages 125–132, 2002.

[150] S. Tenginakai, J. Lee, and R. Machiraju. Iso-surface detection with model-independent statistical signatures. In *Proceedings of IEEE Visualization 2001*, pages 231–238, 2001.

[151] T. Theußl, H. Hauser, and E. Gröller. Mastering windows: Improving reconstruction. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 101–108, 2000.

[152] T. Theußl. *Sampling and Reconstruction in Volume Visualization.* Diplomarbeit, Vienna University of Technology, 1999.

[153] P. Thévenaz, T. Blu, and M. Unser. Interpolation revisited. *IEEE Transactions on Medical imaging*, 19(7):739–758, 2000.

[154] U. Tiede, T. Schiemann, and K.-H. Höhne. High quality rendering of attributed volume data. In *Proceedings of IEEE Visualization '98*, pages 255–262, 1998.

[155] S. Treavett and M. Chen. Pen-and-ink rendering in volume visualisation. In *Proceedings of IEEE Visualization 2000*, pages 203–210, 2000.

[156] K. Turkowski. Filters for common resampling tasks. In A. Glassner, editor, *Graphics Gems I*, pages 147–165. Academic Press, 1990.

[157] G. Turk and J. O'Brien. Shape transformation using variational implicit functions. In *Proceedings of SIGGRAPH '99*, pages 335–342, 1999.

[158] F.-Y. Tzeng, E. Lum, and K.-L. Ma. A novel interface for higher-dimensional classification of volume data. In *Proceedings of IEEE Visualization 2003*, pages 505–512, 2003.

[159] K. Udupa and G. Herman. *3D Imaging in Medicine*. CRC Press, 1999.

[160] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 23–ff., 1996.

[161] J. van Wijk. Image based flow visualization. In *Proceedings of SIGGRAPH 2002*, pages 745–754, 2002.

[162] J. van Wijk. Image based flow visualization for curved surfaces. In *Proceedings of IEEE Visualization 2003*, pages 745–754, 2003.

[163] I. Viola, A. Kanitsar, and E. Gröller. Hardware-based nonlinear filtering and segmentation using high-level shading languages. In *Proceedings of IEEE Visualization 2003*, pages 309–316, 2003.

[164] I. Viola. *Applications of Hardware-Accelerated Filtering in Computer Graphics*. Diplomarbeit, Vienna University of Technology, 2002.

[165] M. Weiler and T. Ertl. Hardware-software-balanced resampling for the interactive visualization of unstructured grids. In *Proceedings of IEEE Visualization 2001*, pages 199–206, 2001.

[166] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, 2003.

[167] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3D textures. In *Proceedings of IEEE VolVis 2000*, pages 7–13, 2000.

[168] D. Weiskopf, K. Engel, and T. Ertl. Volume clipping via per-fragment operations in texture-based volume visualization. In *Proceedings of IEEE Visualization 2002*, pages 93–100, 2002.

[169] D. Weiskopf, K. Engel, and T. Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[170] D. Weiskopf, G. Erlebacher, M. Hopf, and T. Ertl. Hardware-accelerated lagrangian-eulerian texture advection for 2D flow visualization. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2002*, pages 77–84, 2002.

[171] D. Weiskopf and T. Ertl. Dye advection without the blur: A level-set approach for texture-based visualization of unsteady flow. In *Proceedings of Eurographics 2004*, 2004.

[172] D. Weiskopf, M. Hopf, and T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2001*, pages 439–446, 2001.

[173] T. Welsh and K. Mueller. A frequency-sensitive point hierarchy for images and volumes. In *Proceedings of IEEE Visualization 2003*, pages 425–432, 2003.

[174] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, pages 169–178, 1998.

[175] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. In *Proceedings of IEEE Visualization 2001*, pages 271–278, 2001.

[176] R. Westermann. The rendering of unstructured grids revisited. In *Proceedings of Joint Eurographics/IEEE TCVG Symposium on Visualization 2001*, pages 65–74, 2001.

[177] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH '90*, pages 367–376, 1990.

[178] L. Williams. Pyramidal parametrics. In *Proceedings of SIGGRAPH '83*, pages 1–11, 1983.

[179] C. Wittenbrink, T. Malzbender, and M. Goss. Opacity-weighted color interpolation for volume sampling. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 135–142, 1998.

[180] G. Wolberg. *Digital Image Warping.* IEEE Computer Society Press, 1990.

[181] K. Wu. Direct calculation of mip-map level for faster texture mapping. Technical Report HPL-98-112, HP Labs, Computer Systems Laboratory, 1998.

[182] K. Wu. Rational-linear interpolation of texture coordinates and their partial derivatives. Technical Report HPL-98-113, HP Labs, Computer Systems Laboratory, 1998.

[183] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, pages 371–378, 2001.