
Course Notes

Advanced Illumination Techniques for GPU-Based Volume Raycasting

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Patric Ljung

Siemens Corporate Research, Princeton, USA

Christof Rezk Salama

University of Siegen, Germany

Timo Ropinski

University of Münster, Germany



SIGGRAPH2009

Advanced Illumination Techniques for GPU Volume Raycasting

Abstract Volume raycasting techniques are important for both visual arts and visualization. They allow an efficient generation of visual effects and the visualization of scientific data obtained by tomography or numerical simulation. Thanks to their flexibility, experts agree that GPU-based raycasting is the state-of-the-art technique for interactive volume rendering. It will most likely replace existing slice-based techniques in the near future. Volume rendering techniques are also effective for the direct rendering of implicit surfaces used for soft body animation and constructive solid geometry.

The lecture starts off with an in-depth introduction to the concepts behind GPU-based ray-casting to provide a common base for the following parts. The focus of this course is on advanced illumination techniques which approximate the physically-based light transport more convincingly. Such techniques include interactive implementation of soft and hard shadows, ambient occlusion and simple Monte-Carlo based approaches to global illumination including translucency and scattering.

With the proposed techniques, users are able to interactively create convincing images from volumetric data whose visual quality goes far beyond traditional approaches. The optical properties in participating media are defined using the phase function. Many approximations to the physically based light transport applied for rendering natural phenomena such as clouds or smoke assume a rather homogenous phase function model. For rendering volumetric scans on the other hand different phase function models are required to account for both surface-like structures and fuzzy boundaries in the data. Using volume rendering techniques, artists who create medical visualization for science magazines may now work on tomographic scans directly, without the necessity to fall back to

creating polygonal models of anatomical structures.

Course participants should have a working knowledge in computer graphics, basic programming skills. They should be familiar with graphics hardware and shading languages. We will assume a basic knowledge regarding volume data as well as interactive volume rendering techniques.

Prerequisites

Intermediate The course targets the steadily growing number of developers who create specialized implementations of volume rendering techniques on state-of-the-art graphics hardware, regardless of whether they are working in visual arts or scientific visualization.

Level of Difficulty

Contact

Christof Rezk Salama

(course organizer)

Computer Graphics Group

University of Siegen

Hölderlinstr. 3

57068 Siegen, Germany

email: rezk@fb12.uni-siegen.de

Timo Ropinski

Visualization and Computer

Graphics Research Group,

University of Münster

Einsteinstr. 62

48149 Münster, Germany

Markus Hadwiger

VRVis Research Center for

Virtual Reality and Visualization

Donau-City-Straße 1

A-1220 Vienna, Austria

email: msh@vrvis.at

Patric Ljung

Siemens Corporate Research

Imaging & Visualization Department

755 College Road East

Princeton, NJ 08540

email: patric.ljung@siemens.com

Lecturers

Markus Hadwiger

VRVis Research Center for Virtual Reality and Visualization
Donau-City-Strasse 1,
A-1220 Vienna, Austria
email: msh@vrvis.at

Markus Hadwiger is a senior researcher at the VRVis Research Center in Vienna, Austria. He received his Ph.D. in computer science from the Vienna University of Technology in 2004, and has been a researcher at VRVis since 2000, working in the areas of visualization, volume rendering, and general GPU techniques. He has been involved in several courses and tutorials about volume rendering and visualization at ACM SIGGRAPH, IEEE Visualization, and Eurographics. He is a co-author of the book *Real-Time Volume Graphics* published by A K Peters.

Patric Ljung

Department of Imaging and Visualization
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540, U.S.A.
email: patric.ljung@siemens.com

Patric Ljung joined in 2007 Siemens Corporate Research in Princeton, NJ, where he works as a Research Scientist in the Imaging Architectures group. He received 2006 his PhD in Scientific Visualization from Linköping University, Sweden and graduated with honors in 2000 his MS in Information Technology from Linköping University. Between 1989 and 1995 he worked as a software engineer with embedded and telecom systems involving software architectures, graphical user interfaces, voice-mail systems, communication protocols, network and interprocess communication, compilers.

Dr. Ljung has published several papers in international conferences and journals including IEEE Visualization, Eurographics conferences, IEEE TVCG and others, on volume rendering of large medical data sets, GPU-based raycasting of multiresolution data sets. One important focus area has been Virtual Autopsies for forensic pathology. His current research interest is in advanced illumination and shading techniques, software architectures for extensible graphics, and management and

rendering of large medical data sets.

Timo Ropinski

Visualization and Computer Graphics Research Group (VisCG)
University of Münster
Einsteinstr. 62
48149 Münster, Germany
email: ropinski@math.uni-muenster.de

Timo Ropinski is a postdoctoral researcher working in the field of medical volume visualization. After receiving his PhD in 2004 from the University of Münster, he became a project leader within the collaborative research center SFB 656, a cooperation between researchers from medicine, mathematics, chemistry, physics and computer science. His research is focused on interactive aspects in medical volume visualization with the goal to make these techniques more accessible. He is initiator of the Voreen open source project (www.voreen.org), in which a flexible volume rendering framework is developed. The results of his scientific work have been published in various international conferences including Eurographics, IEEE Visualization, IEEE VR, VMV and others.

Christof Rezk Salama

Computergraphik und Multimediasysteme,
University of Siegen,
Hölderlinstr. 3,
57068 Siegen, Germany
phone: +49 271-740-3315
fax: +49 271-740-3337
email: rezk@fb12.uni-siegen.de

Christof Rezk-Salama has received a PhD from the University of Erlangen-Nuremberg as a scholarship holder of the graduate college *3D Image Analysis and Synthesis*. He has worked as a research engineer for the R&D department of Siemens Medical Solutions. Since October 2003 he is working as an assistant professor at the Computer Graphics Group of the University of Siegen, Germany.

The results of his research have been presented at international conferences, including ACM SIGGRAPH, IEEE Visualization, Eurographics, MICCAI and Graphics Hardware. He is regularly holding lectures, courses and seminars on computer graphics, scientific visualization, char-

acter animation and graphics programming.

He has gained practical experience in applying computer graphics to several scientific projects in medicine, geology and archaeology. Christof Rezk-Salama has released the award winning open-source project *OpenQVis* and is a co-author of the book *Real-Time Volume Graphics*.

Detailed information about this research projects are available at:

<http://www.cg.informatik.uni-siegen.de/People/Rezk>

<http://www.real-time-volume-graphics.org>

<http://openqvis.sourceforge.net>

Course Syllabus

The half-day course will consist of four different blocks. From a didactic point of view, each block will loosely build upon the information provided in previous blocks with growing complexity and increasing level of difficulty.

The schedule is only tentative, since at the time of writing these notes, the final time slots have not yet been allocated by the organizers.

MORNING

Introduction and Basics [45 min] (M. Hadwiger)

- Introduction and Basics
- Application Areas for Volume Rendering
- Benefits and Drawbacks of Ray-Casting
- GPU-based Volume Ray-Casting
- Space Leaping and Early Ray Termination
- Memory Management
- Multiresolution LOD and Adaptive sampling

**1:45pm –
2:30pm**

Light Interaction [45 min] (T. Ropinski)

- Light Transport and Illumination Models
- Local Volume Illumination
- Specular Reflection through Ray-Tracing
- Soft vs. Hard Shadows
- Semi-Transparent Shadows with Deep Shadow Maps

**2:30pm –
3:15pm**

BREAK [15 min]

**3:15pm –
3:30pm**

3:30pm – Ambient Occlusion [45 min] (P. Ljung)

- 4:15pm**
- Ambient Occlusion for Isosurfaces
 - Deep Shadow Maps
 - Local Ambient Occlusion (DVR)
 - Dynamic Ambient Occlusion (DVR)

4:15pm- Scattering [60 min] (C. Rezk-Salama)

- 5:15pm**
- Single and Multiple Scattering
 - Transparency and Translucency
 - Monte-Carlo integration
 - GPU-Based Importance Sampling
 - GPU-Based Monte-Carlo Volume Raycasting
 - Scattering with Deep Shadow Maps

5:15pm- Discussion, Questions and Answers [15min] (all speakers)

5:30pm

Contents

I GPU-Based Ray Casting	1
1 Introduction	2
1.1 Volume Data	3
1.2 Direct Volume Rendering	4
1.2.1 Optical Models	5
1.2.2 The Volume Rendering Integral	6
1.2.3 Ray Casting	8
1.2.4 Alpha Blending	10
2 GPU-based Volume Ray Casting	12
2.1 Basic Ray Casting	13
2.2 Object-Order Empty Space Skipping	15
2.3 Advanced Ray Casting Pipeline	17
2.3.1 Culling and Brick Boundary Rasterization	20
2.3.2 Geometry Intersection	23
2.4 Isosurface Ray Casting	24
2.4.1 Adaptive Sampling	25
2.4.2 Intersection Refinement	28
2.5 Memory Management	28
2.6 Mixed-Resolution Volume Rendering	30
2.6.1 Volume Subdivision for Texture Packing	31
2.6.2 Mixed-Resolution Texture Packing	32
2.6.3 Address Translation	33
2.7 Multiresolution LOD and Adaptive sampling	35
2.7.1 Octree-based Multiresolution Representation	35
2.7.2 Block Properties and Acceleration Structures	36
2.7.3 Hierarchical Multiresolution Representations	37
2.8 Level-of-Detail Management	38
2.8.1 View-Dependent Approaches	39
2.8.2 Data Error Based Approaches	39

2.8.3	Transfer Function Based Approaches	40
2.9	Encoding, Decoding and Storage	40
2.9.1	Transform and Compression Based Techniques	41
2.9.2	Out-of-Core Data Management Techniques	43
2.9.3	Flat blocking Multiresolution Representation	44
2.10	Sampling of Multiresolution Volumes	46
2.10.1	Nearest Block Sampling	47
2.10.2	Interblock Interpolation Sampling	48
2.10.3	Interblock Interpolation Results	50
2.11	Raycasting on the GPU	51
2.11.1	Adaptive Object-Space Sampling	51
2.11.2	Flat Blocking Summary	53
II	Light Interaction	55
3	Light Transport and Illumination Models	56
3.1	Phong Illumination	56
3.2	Gradient Computation	59
3.3	Specular Reflections through Ray-Tracing	61
4	Shadows	68
4.1	Soft vs. Hard Shadows	68
4.2	Semi-Transparent Shadows with Deep Shadow Maps	70
III	Ambient Occlusion	77
5	Ambient Occlusion for Isosurfaces	78
6	Ambient Occlusion for Direct Volume Rendering	80
6.1	Local Ambient Occlusion	80
6.1.1	Emissive Tissues and Local Ambient Occlusion	82
6.1.2	Integrating Multiresolution Volumes	82
6.1.3	Adding Global Light Propagation	84
6.2	Dynamic Ambient Occlusion	85
6.2.1	Local Histogram Generation	87
IV	Volume Scattering	99
7	Scattering Effects	100
7.1	Physical Background	100
7.2	Scattering	101
7.3	Single Scattering	101

7.4	Indirect Illumination and Multiple Scattering	103
7.4.1	Indirect Light	103
7.4.2	Transparency and Translucency	104
7.4.3	Phase Functions	105
7.4.4	Scattering at Transparent Surfaces	106
7.5	A Practical Phase Function Model	108
7.6	Further Reading	109
8	Monte-Carlo Intergration	110
8.1	Numerical Integration	110
8.1.1	Blind Monte-Carlo Integration	110
8.2	When Does Monte-Carlo Integration Make Sense?	112
8.3	Importance Sampling	114
8.4	GPU-based Importance Sampling	116
8.4.1	Focussing of Uniform Distribution	116
8.4.2	Sampling of Reflection MIP-Maps	118
8.5	Further Reading	121
9	GPU-Based Monte-Carlo Volume Raycasting	123
9.1	Monte-Carlo Techniques for Isosurfaces	123
9.2	Isosurfaces with Shift-Variant or Anisotropic BRDFs	125
9.2.1	First Hit Pass	125
9.2.2	Deferred Shading Pass	128
9.2.3	Deferred Ambient Occlusion Pass	130
9.3	Volume Scattering	133
9.3.1	Heuristic Simplifications	137
10	Light Map Approaches	140

Course Notes

Advanced Illumination Techniques for GPU Volume Raycasting

GPU-Based Ray Casting

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Patric Ljung

Siemens Corporate Research, Princeton, USA

Christof Rezk Salama

University of Siegen, Germany

Timo Ropinski

University of Münster, Germany



SIGGRAPH2009

Introduction

In traditional modeling, 3D objects are created using surface representations such as polygonal meshes, NURBS patches or subdivision surfaces. In the traditional modeling paradigm, visual properties of surfaces, such as color, roughness and reflectance, are modeled by means of a shading algorithm, which might be as simple as the Phong model or as complex as a fully-featured shift-variant anisotropic BRDF. Since light transport is evaluated only at points on the surface, these methods usually lack the ability to account for light interaction which is taking place in the atmosphere or in the interior of an object.

Contrary to surface rendering, volume rendering [60, 23] describes a wide range of techniques for generating images from three-dimensional scalar data. These techniques are originally motivated by scientific visualization, where volume data is acquired by measurement or numerical simulation of natural phenomena. Typical examples are medical data of the interior of the human body obtained by computed tomography (CT) or magnetic resonance imaging (MRI). Other examples are computational fluid dynamics (CFD), geological and seismic data, as well as abstract mathematical data such as 3D probability distributions of pseudo random numbers.

With the evolution of efficient volume rendering techniques, volumetric data is becoming more and more important also for visual arts and computer games. Volume data is ideal to describe fuzzy objects, such as fluids, gases and natural phenomena like clouds, fog, and fire. Many artists and researchers have generated volume data synthetically to supplement surface models, i.e., procedurally [24], which is especially useful for rendering high-quality special effects.

Although volumetric data are more difficult to visualize than surfaces, it is both worthwhile and rewarding to render them as truly three-dimensional entities without falling back to 2D subsets.

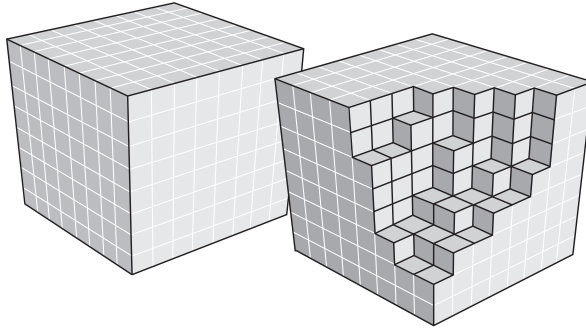


Figure 1.1: Voxels constituting a volumetric object after it has been discretized.

1.1 Volume Data

A discrete volume data set can be thought of as a simple three-dimensional array of cubic elements (voxels¹) [49], each representing a unit of space (Figure 1.1).

Although imagining voxels as tiny cubes is easy and might help to visualize the immediate vicinity of individual voxels, it is more appropriate to identify each voxel with a sample obtained at a single infinitesimally small point from a continuous three-dimensional signal

$$f(\mathbf{x}) \in \mathbb{R} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^3. \quad (1.1)$$

Provided that the continuous signal is band-limited with a cut-off-frequency ν_s , sampling theory allows the exact reconstruction, if the signal is evenly sampled at more than twice the cut-off-frequency, i.e., the Nyquist rate. However, there are two major problems which prohibit the ideal reconstruction of sampled volume data in practice.

- Ideal reconstruction according to sampling theory requires the convolution of the sample points with a *sinc* function (Figure 1.2a) in the spatial domain. For the one-dimensional case, the sinc function is:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}. \quad (1.2)$$

The three-dimensional version of this function is simply obtained by tensor-product. Note that this function has infinite extent. Thus, for an exact reconstruction of the original signal at an arbitrary position *all* the sampling points must be considered, not only

¹volume elements

those in a local neighborhood. This turns out to be computationally intractable in practice.

- Real-life data in general does not represent a band-limited signal. Any sharp boundary between different materials represents a step function which has infinite extent in the frequency domain. Sampling and reconstruction of a signal which is not band-limited will produce aliasing artifacts.

In order to reconstruct a continuous signal from an array of voxels in practice, the ideal 3D *sinc* filter is usually replaced by either a box filter (Figure 1.2a) or a tent filter (Figure 1.2b). The box filter calculates nearest-neighbor interpolation, which results in sharp discontinuities between neighboring cells and a rather blocky appearance. Trilinear interpolation, which is achieved by convolution with a 3D tent filter, represents a good trade-off between computational cost and smoothness of the output signal.

1.2 Direct Volume Rendering

In comparison to the indirect methods, which try to extract a surface description from the volume data in a preprocessing step, direct methods display the voxel data by evaluating an *optical model* which describes how the volume emits, reflects, scatters, absorbs and occludes light [73]. The scalar value is virtually mapped to physical quantities which describe light interaction at the respective point in 3D space. This mapping is often called *classification* and is usually performed by means of a *transfer function*. The physical quantities are then used for images synthesis.

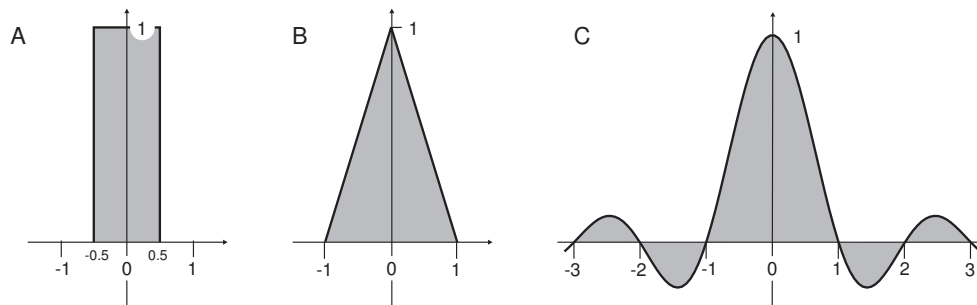


Figure 1.2: Reconstruction filters for one-dimensional signals. In practice, box filter (A) and tent filter (B) are used instead of the ideal *sinc*-filter (C).

Different optical models for direct volume rendering are described in section 1.2.1.

During image synthesis, the light propagation is computed by integrating light interaction effects along viewing rays based on the optical model. The corresponding integral is known as the *volume rendering integral*, which is described in section 1.2.2. Naturally, under real-world conditions this integral is solved numerically. Furthermore, the volume can be shaded according to the *illumination* from external light sources.

1.2.1 Optical Models

Almost every direct volume rendering algorithm regards the volume as a distribution of light-emitting particles of a certain density. These densities are more or less directly mapped to RGBA quadruplets for compositing along viewing rays. This procedure, however, is motivated by a physically-based optical model.

The most important optical models for direct volume rendering are described in a survey paper by Nelson Max [73], 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 that only emit light, but do not absorb any, since the absorption is negligible.
- **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 before it is scattered toward the eye.

The *volume rendering integral* described in the following section assumes the simple emission-absorption optical model. More sophisticated models including shadowing and self-shadowing, and single and multiple scattering are covered in later parts of these notes.

Figure 1.3 illustrates GPU-based ray casting with the emission-absorption model with and without shading, as well as a combination with semi-transparent isosurface rendering. Figure 1.4 illustrates the addition of shadows, i.e., the (partial) occlusion of impinging external light via the absorption occurring within the volume.

1.2.2 The Volume Rendering Integral

Every physically-based volume rendering algorithm evaluates the volume rendering integral in one way or the other, even if viewing rays are not employed explicitly by the algorithm. The most basic, but also most flexible, volume rendering algorithm is ray casting, which is introduced in Section 1.2.3. It might be considered as the “most direct” numerical method for evaluating this integral. More details are covered later on, but for this section 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.

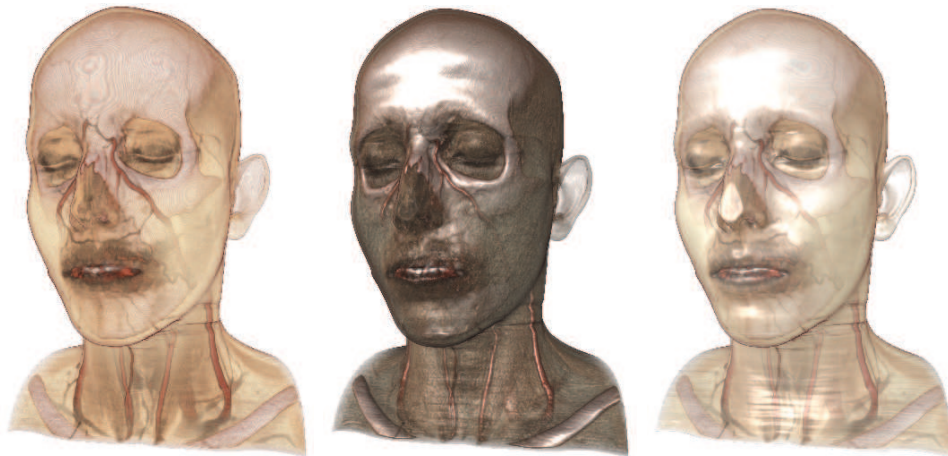


Figure 1.3: Direct volume rendering with emission-absorption (left); plus shading (center); combined with a shaded semi-transparent isosurface (right).

Note that this general description assumes both the volume and the mapping to optical properties to be continuous. In practice, of course, the volume data are discrete, and the evaluation of the integral is approximated numerically. In combination with several additional simplifications, the integral is usually substituted by a Riemann sum.

We denote a ray cast into the volume by $\mathbf{x}(t)$, and parameterize it by the distance t from the eye. The scalar value corresponding to a position along the ray is denoted by $s(\mathbf{x}(t))$. If we employ the emission-absorption model, the volume rendering equation integrates *absorption coefficients* $\kappa(s)$ (accounting for the absorption of light), and *emissive colors* $c(s)$ (accounting for radiant energy actively emitted) along a ray. To keep the equations simple, we denote emission c and absorption coefficients κ as function of the eye distance t instead of the scalar value s :

$$c(t) := c(s(\mathbf{x}(t))) \quad \text{and} \quad \kappa(t) := \kappa(s(\mathbf{x}(t))) \quad (1.3)$$

Figure 1.5 illustrates the idea of emission and absorption. An amount of radiant energy, which is emitted at a distance $t = d$ along the viewing ray is continuously absorbed along the distance d until it reaches the eye. This means that only a portion c' of the original radiant energy c emitted



Figure 1.4: Rendering of a CT scan of a human head (512x512x333) with direct volume rendering and shadowing with GPU-accelerated deep shadow maps. The shadow map resolution is 512x512. Both volume rendering and construction of the deep shadow map are performed by ray casting on the GPU [39].

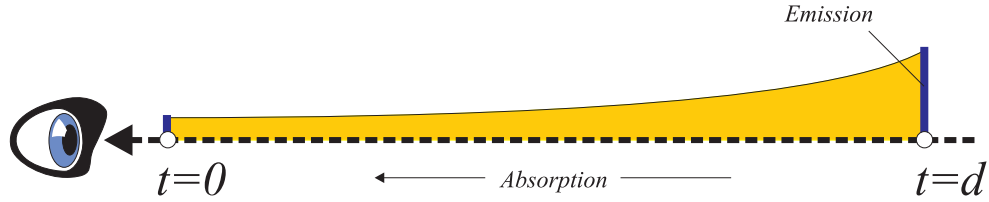


Figure 1.5: An amount of radiant energy emitted at $t = d$ is partially absorbed along the distance d .

at $t = d$ will eventually reach the eye. If there is a constant absorption $\kappa = const$ along the ray, c' amounts to

$$c' = c \cdot e^{-\kappa d} \quad . \quad (1.4)$$

However, if the absorption κ is not constant along the ray, but itself dependent on the position, the amount of radiant energy c' reaching the eye must be computed by integrating the absorption coefficient along the distance d :

$$c' = c \cdot e^{-\int_0^d \kappa(\hat{t}) d\hat{t}} \quad . \quad (1.5)$$

The integral over the absorption coefficients in the exponent,

$$\tau(d_1, d_2) = \int_{d_1}^{d_2} \kappa(\hat{t}) d\hat{t} \quad (1.6)$$

is also called the *optical depth*. In this simple example, however, light was only emitted at a single point along the ray. If we want to determine the total amount of radiant energy C reaching the eye from this direction, we must take into account the emitted radiant energy from all possible positions t along the ray:

$$C = \int_0^\infty c(t) \cdot e^{-\tau(0, t)} dt \quad (1.7)$$

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

1.2.3 Ray Casting

Ray casting [60] is an image-order direct volume rendering algorithm, which uses straight-forward numerical evaluation of the volume render-

ing integral (Equation 1.7). For each pixel of the image, a single ray² is cast into the scene. At equi-spaced intervals along the ray, the discrete volume data are 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 quadruplet that subsumes the corresponding emission and absorption coefficients [60] for this location. The solution of the volume rendering integral is then approximated via alpha blending in either front-to-back or back-to-front order, where usually the former is used in ray casting.

The optical depth τ (Equation 1.6), which is the cumulative absorption up to a certain position $\mathbf{x}(t)$ along the ray, can be approximated by a Riemann sum

$$\tau(0, t) \approx \tilde{\tau}(0, t) = \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t) \Delta t \quad (1.8)$$

with Δt denoting the distance between successive resampling locations. The summation in the exponent can immediately be substituted by a multiplication of exponentiation terms:

$$e^{-\tilde{\tau}(0, t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} e^{-\kappa(i \cdot \Delta t) \Delta t} \quad (1.9)$$

Now, we can introduce the *opacity* A , which is well-known from traditional alpha blending, by defining

$$A_i = 1 - e^{-\kappa(i \cdot \Delta t) \Delta t} \quad (1.10)$$

and rewriting Equation 1.9 as:

$$e^{-\tilde{\tau}(0, t)} = \prod_{i=0}^{\lfloor t/d \rfloor} (1 - A_j) \quad (1.11)$$

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

Similarly, the emitted color of the i -th ray segment can be approximated by:

$$C_i = c(i \cdot \Delta t) \Delta t \quad (1.12)$$

²assuming super-sampling is not used for anti-aliasing

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 $n = \lfloor T/\delta t \rfloor$)

$$\tilde{C} = \sum_{i=0}^n C_i \prod_{j=0}^{i-1} (1 - A_j) \quad (1.13)$$

Equation 1.13 can be evaluated iteratively by performing *alpha blending* in either front-to-back or back-to-front order.

1.2.4 Alpha Blending

Equation 1.13 can be computed iteratively in front-to-back order by stepping i from 1 to n :

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \quad (1.14)$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \quad (1.15)$$

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 in all blending equations, we are using *opacity-weighted colors* [110], which are also known as *associated colors* [7]. 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 [110].

The following alternative iterative formulation evaluates Equation 1.13 in back-to-front order by stepping i from $n - 1$ to 0:

$$C'_i = C_i + (1 - A_i)C'_{i+1} \quad (1.16)$$

A new value C'_i is calculated from the color C_i and opacity A_i at the current location i , and the composite color C'_{i+1} from the previous location $i + 1$. The starting condition is $C'_n = 0$.

Note that front-to-back compositing requires tracking alpha values, whereas back-to-front compositing does not. However, while this was a problem for hardware implementations several years ago, in current single-pass implementations of GPU ray casting this is not a problem at

all. In multi-pass implementations, *destination alpha* must be supported by the frame buffer for tracking the accumulation of opacity, i.e., an alpha value must be stored in the frame buffer, and it must be possible to use it as a multiplication factor in blending operations.

The major advantage of front-to-back compositing is an optimization called *early ray termination*, where the progression along a ray is terminated as soon as the cumulative alpha value reaches 1.0, or a sufficiently close value. In current GPU architectures, this is very easy to implement by simply terminating the ray casting loop as soon as the accumulated alpha value exceeds a specified threshold.

GPU-based Volume Ray Casting

The basic idea of GPU-based ray casting is to store the entire volume in a single 3D texture, and drive a fragment program that casts rays into the volume. Each pixel/fragment corresponds to a single ray $\mathbf{x}(t, x, y) = \mathbf{c} + t \mathbf{d}(x, y)$ in volume coordinates. Here, the normalized direction vector $\mathbf{d}(x, y)$ can either be computed from the camera position \mathbf{c} and the screen space coordinates (x, y) of the pixel, or be obtained via rasterization [55]. In this section, we will use the approach building on rasterization since it allows for very simple but efficient empty space skipping, which is described in later sections. The range of depths $[t_{start}(x, y), t_{exit}(x, y)]$ from where a ray enters the volume to where a ray exits the volume is computed per frame in a setup stage before the actual ray casting fragment program is executed. In the simplest case, t_{start} , or the corresponding 3D volume coordinates, are obtained by rasterizing the front faces of the volume bounding box with the corresponding distance to the camera. Rendering the back faces of the bounding box yields the depths t_{exit} , or the corresponding 3D volume coordinates, of each ray exiting the volume.

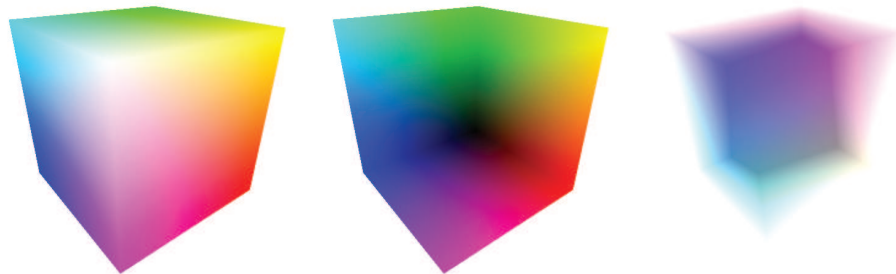


Figure 2.1: Rasterization for ray setup. The back face coordinates (center), minus the front face coordinates (left) yield ray direction vectors and lengths (right). 3D volume coordinates in $[0, 1]^3$ are illustrated as RGB colors, i.e., the entire RGB color cube corresponds to the volume bounding box.

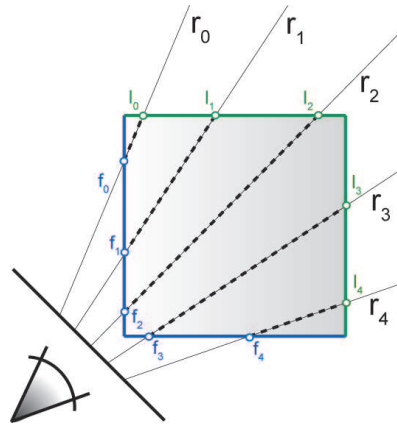


Figure 2.2: In the ray casting pass, the volume is sampled at regular intervals between the starting (f_0 - f_4) and ending (l_0 - l_4) positions obtained via rasterization.

Figure 2.1 illustrates this ray setup using rasterization. As illustrated in Figure 2.2, ray entry positions are determined by the front faces of the volume bounding box (shown in blue), and ray exit positions by its back faces (shown in green), respectively. Ray casting is performed by sampling the space in-between, usually by using a constant sampling rate. On current GPUs, a single rendering pass and ray casting loop in the fragment program can be employed for casting through the volume in front-to-back order, building on the images illustrated in Figure 2.1 for ray setup, which yield exactly the setup positions needed by the ray caster (f_0 - f_4 , and l_0 - l_4 in Figure 2.2).

2.1 Basic Ray Casting

Figure 2.3 illustrates basic ray casting with ray setup using rasterization. It consists of four principal stages:

1. Front face generation: Render the front faces of the volume bounding box to a texture (Figure 2.1 (left)).
2. Direction texture generation: Render the back faces of the volume bounding box (Figure 2.1 (center)), while subtracting the previously generated coordinates of the front faces and storing the resulting ray vectors as normalized vectors in RGB, as well as their lengths in A, of a separate RGBA direction texture (Figure 2.1 (right)).

3. Ray casting: Get the starting position from the front face image and cast along the viewing vector until the ray has left the volume. Exiting the volume is determined by using the previously stored vector lengths.
4. Blending: Blend the ray casting result to the screen, e.g., composite it with the background.

The only expensive stage of this algorithm is the actual ray casting loop, which iteratively steps through the volume, sampling the 3D volume texture using tri-linear interpolation, applies the transfer function, and performs compositing. Ray setup via rasterization is several orders of magnitude faster with negligible performance impact, and thus no bottleneck. The final blending stage is negligible in terms of performance as well, or can even be skipped entirely if the ray casting pass is executed directly on the final output buffer.

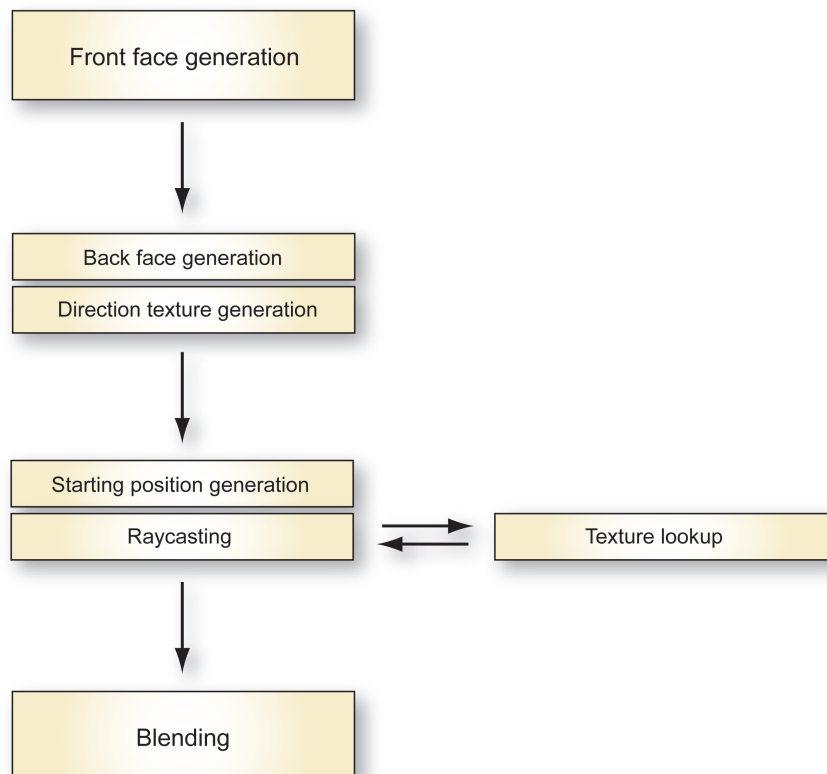


Figure 2.3: The rendering pipeline of the basic GPU ray casting algorithm.

2.2 Object-Order Empty Space Skipping

When we consider Figure 2.2, and imagine that the actually visible part of the volume does not fill up the entire bounding box, we see that a lot of *empty space* will be sampled during ray casting if rays are started on the front faces of the volume bounding box. However, if we subdivide the volume into smaller blocks and determine for each of these blocks whether it is empty or not, we can rasterize the front faces of these smaller blocks instead of the entire bounding box. This can simply be achieved by rasterizing front and back faces of smaller blocks, resulting in ray setup images as shown in Figure 2.4, which already more closely resemble the visible part of the volume (in the case of this figure, a human skull and spine). This is illustrated in Figure 2.5, where both the ray entry positions (f_0 - f_2) as well as the ray exit positions (l_0 - l_4) have been modified via this rasterization of block bounding faces to be inside the volume bounding box and closer to the visible part of the volume.

Figure 2.6 illustrates a potential performance problem of this approach, which occurs when rays graze the volume early on, but do not hit a visible part right away (right-hand side of the figure). In this case, a lot of empty space may be traversed. However, this case usually occurs only for a small number of rays, and can be handled by combining object-order empty space skipping with regular (image-order) empty space skipping, i.e., deciding in the ray casting fragment program to skip individual samples or advancing the sampling position along the ray by several samples

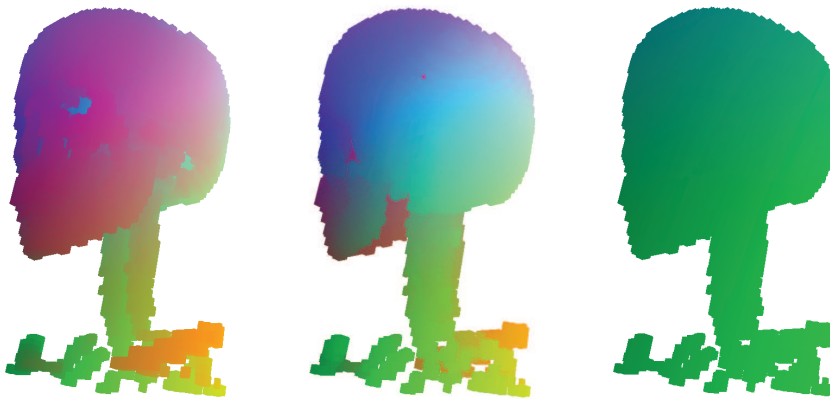


Figure 2.4: Geometry setup for ray casting with object-order empty space skipping. The complexity of the bounding geometry is adapted to the underlying dataset.

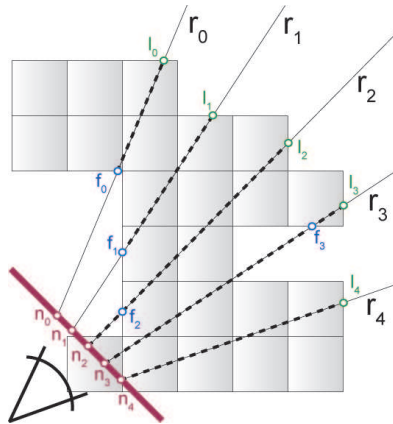


Figure 2.5: Determining ray start positions and ray lengths using rasterization of the faces of a tightly fitting bounding geometry for object-order empty space skipping.

at once. These two approaches for empty space skipping complement each other. Object-order empty space skipping is extremely fast (with negligible overhead compared with no empty space skipping), it employs very simple and fast rasterization, and the ray casting fragment program does not need to be modified at all. It, however, in principle cannot skip all empty space. Image-order empty space skipping, on the other

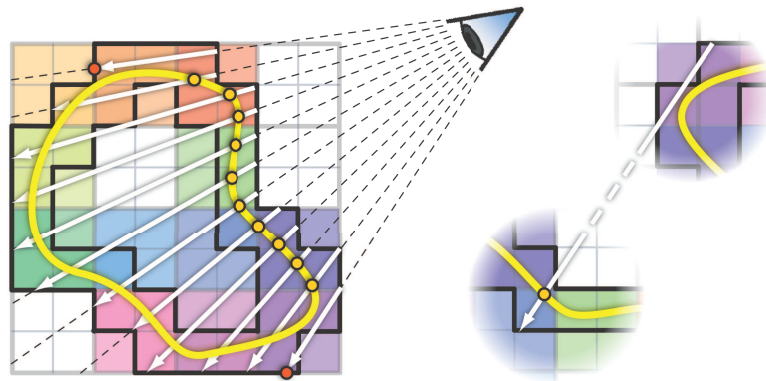


Figure 2.6: Ray casting with object-order empty space skipping. The bounding geometry (black) between active and inactive blocks that determines start and exit depths for sampling along rays (white) encloses an isosurface (yellow), in this example. Actual ray termination points are shown in yellow and red, respectively.

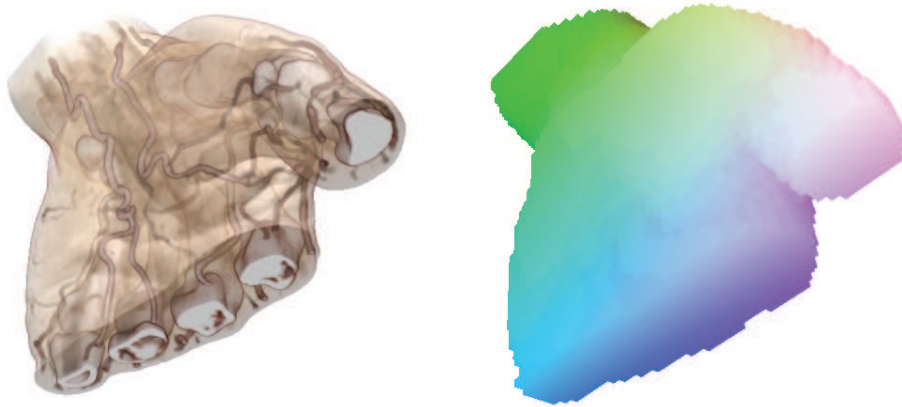


Figure 2.7: Replacing the simple volume bounding box with a tighter bounding geometry implicitly skips all of the outer empty space (in front and behind the visible part of the volume) at almost no cost.

hand, either requires multiple ray casting passes or must perform checks on essentially a per-sample basis, and thus is much more expensive. It, however, can skip additional empty space that would otherwise be sampled. Figure 2.7 illustrates another example of object-order empty space skipping via rasterization of tight-fitting bounding geometry.

Figure 2.5 illustrates another important issue of ray setup, which is the handling of rays when the view point is inside the volume. Rays r_3 and r_4 in this figure cannot be started on front faces of bounding geometry, because they have to start inside it, i.e., on the near plane of the view frustum (positions n_3 and n_4). The next section describes an advanced ray casting pipeline that correctly handles this case, as well as the intersection of the volume with opaque geometry, e.g., navigational markers or tools in medical interventions.

2.3 Advanced Ray Casting Pipeline

This section describes an advanced ray casting pipeline that combines object-order and image-order stages in order to find a balance between the two, and leverage the parallel processing of modern GPUs [90]. For culling of irrelevant subvolumes, a regular grid of min-max values for bricks of size 8^3 is stored along with the volume. Ray casting itself is

performed in a single rendering pass in order to avoid the setup overhead of casting each brick separately [44]. The first step of the algorithm culls bricks on the CPU and generates a bit array that determines whether a brick is *active* or *inactive*. This bit array contains the state of bricks with respect to the active parts of the volume, where a brick is active when it contains samples that are mapped to opacities greater than zero by the transfer function and inactive otherwise.

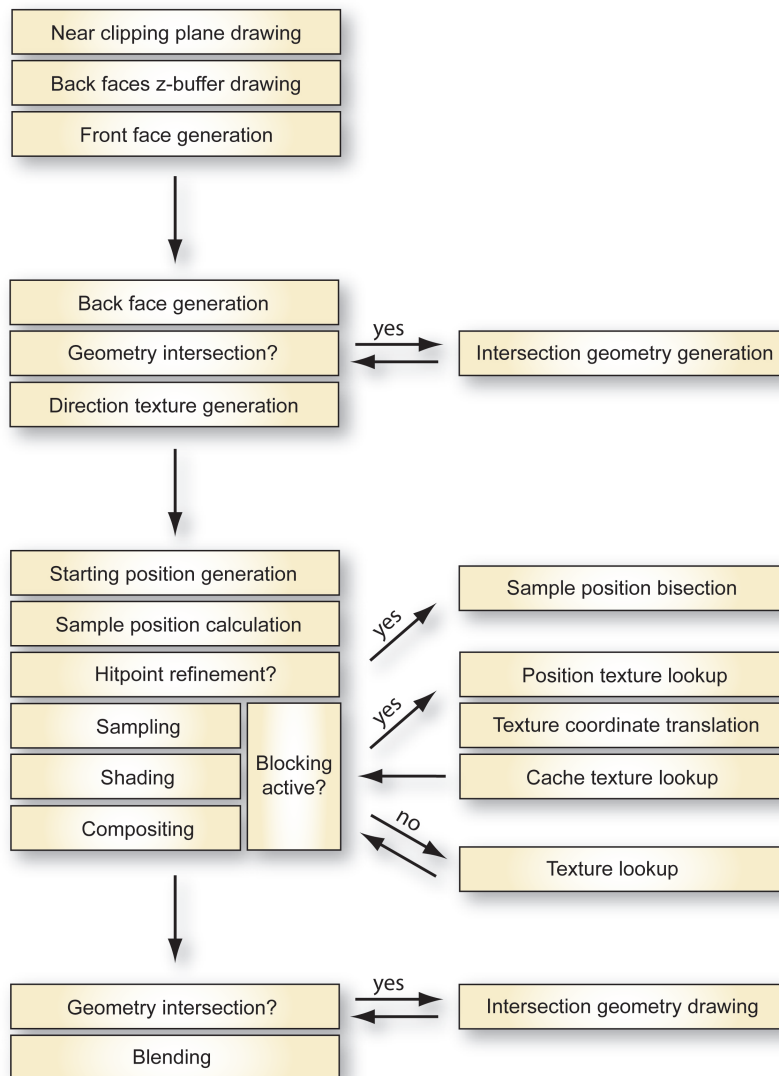


Figure 2.8: The pipeline of the advanced GPU ray caster.

In the object-order stage on the GPU, the bit array is used to rasterize brick boundary faces in several rendering passes. The result of these rendering passes are two images that drive the subsequent ray casting stage. The first image, the *ray start position image*, contains the volume coordinate positions where ray casting should start for each pixel. Coordinates are stored in the RGB components, and the alpha (A) component is one when a ray should be started, and zero when no ray should be started. The second image, the *ray length image* contains the direction vectors for ray casting in the RGB components and the length of each ray in the alpha component. Note that the direction vectors could easily be computed in the fragment program from the camera position and the ray start positions as well. However, the ray length must be rendered into an image that is separate from the ray start positions due to read-write dependencies, which can then also be used for storing the direction vectors that are needed for ray length computation anyway. The main steps of our ray casting approach for each pixel are:

1. Compute the initial ray start position on the near clipping plane of the current viewport. When the start position is in an inactive brick, calculate the nearest intersection point with the boundary faces of active bricks, in order to skip empty space. The result is stored in the *ray start position image*.
2. Compute the ray length until the last intersection point with boundary faces of bricks that are active. The result is stored in the *ray length image*.
3. Optionally render opaque polygonal geometry and overwrite the ray length image where the distance between the ray start position and the geometry position is less than the stored ray length.
4. Cast from the start position stored in the *ray start position image* along the direction vector until the accumulated opacity reaches a specified threshold (*early ray termination*) or the ray length given by the *ray length image* is exceeded. The result of ray casting is stored in a separate compositing buffer.
5. Blend the ray casting compositing buffer on top of the polygonal geometry.

The two main acceleration schemes exploited here are *object-order empty space skipping* and *early ray termination*. For the former, view-independent culling of bricks and rasterization of their boundary faces are employed, whereas the latter is handled during ray casting.

2.3.1 Culling and Brick Boundary Rasterization

Each brick in the subdivision of the volume is either inactive or active with respect to the transfer function. In order to determine ray start positions and ray lengths, we employ rasterization of the boundary faces between active and inactive bricks, which is illustrated in Figure 2.5. To handle brick culling efficiently, the minimum and maximum voxel values of each brick are stored along with the volume, which are compared at run-time with the transfer function. A brick can be safely discarded when the opacity is always zero between those two values, which can be determined very quickly using summed area tables [30].

Rasterizing the boundary faces between active and inactive bricks results in object-order empty space skipping. It prunes the rays used in the ray casting pass and implicitly excludes most inactive bricks. Note, however, that this approach does not exclude all empty space from ray casting, which can be seen for ray r_3 in Figure 2.5 (left). This is a trade-off that enables ray casting without any per-brick setup overhead and works extremely well in practice.

The border between active and inactive bricks defines a surface that can be rendered as standard OpenGL geometry with the corresponding position in volume coordinates encoded in the RGB colors. All vertices of brick bounding geometry are constantly kept in video memory. Only an additional index array referencing the vertices of active boundary faces have to be updated every time the transfer function changes.

As long as the near clipping plane does not intersect the bounding geometry, rays can always be started at the brick boundary front faces. However, if such an intersection occurs, it will produce holes in the front-

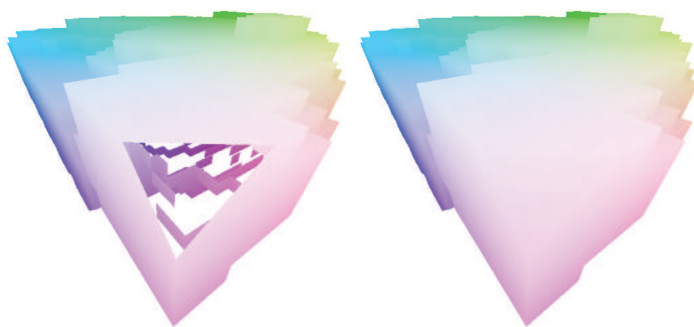


Figure 2.9: Holes resulting from near clipping plane intersection (left) must be filled with valid starting positions (right).

facing geometry, which results in some rays not being started at all, and others started at incorrect positions. Figure 2.9 illustrates this problem. In an endoscopic view, we constantly face this situation, so rays typically need to be started at the near clipping plane, which is shown in Figure 2.5 in the case of points n_2 - n_4 .

To avoid casting through empty space, rays should not be started at the near clipping plane if the starting position is in an inactive brick but at the next intersection with active boundary faces, such as rays r_0 and r_1 in Figure 2.5. These rays are started at f_0 and f_1 , instead of being started at n_0 and n_1 . We achieve this by drawing the near clipping plane first and the front faces afterwards, which ensures that whenever there are no front faces to start from, the position of the near clipping plane will be taken. However, since the non-convex bounding geometry often leads to multiple front faces for a single pixel, the next front face is used when the first front face is clipped, which results in incorrect ray start positions. The solution is to detect when a ray intersects a back face before the first front face that is not clipped.

The basic steps to obtain the *ray start position image* are as follows:

1. Disable depth buffering. Rasterize the entire near clipping plane into the color buffer. Set the alpha channel to zero everywhere.
2. Enable depth buffering. Disable writing to the RGB components of the color buffer. Rasterize the *nearest back faces* of all active bricks into the depth buffer, e.g., by using a depth test of `GL_LESS`. Set the alpha channel to one where fragments are generated.
3. Enable writing to the RGB components of the color buffer. Rasterize the *nearest front faces* of all active bricks, e.g., by once again using a depth test of `GL_LESS`. Set the alpha channel to one where fragments are generated.

This ensures that all possible combinations shown in Figure 2.5 (left) are handled correctly. Rasterizing the nearest front faces makes sure that all near plane positions in inactive bricks will be overwritten by start positions on active bricks that are farther away (rays r_0 and r_1). Rasterizing the nearest back faces before the front faces ensures that near plane positions inside active blocks will not be overwritten by front faces that are farther away (rays r_2 and r_3).

Brick geometry that is nearer than the near clipping plane is automatically clipped by the graphics subsystem. After that, the *ray length*

image can be computed, which first of all means finding the last intersection points of rays with the bounding geometry. The basic steps are:

1. Rasterize the *farthest back faces*, e.g., by using a depth test of `GL_GREATER`.
2. During this rasterization, sample the ray start position image and subtract it from the back positions obtained via rasterization of the back faces. This yields the ray vectors and the ray lengths from start to end position.
3. Multiply all ray lengths with the alpha channel of the ray start position image (which is either 1 or 0).

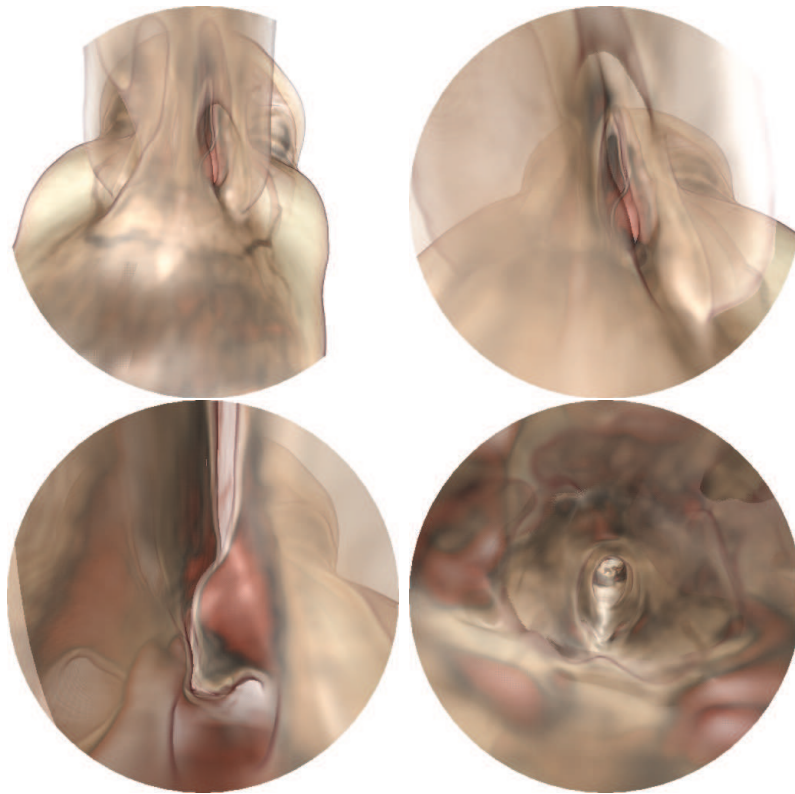


Figure 2.10: Moving the viewpoint inside the volume is especially important for Virtual Endoscopy applications. This sequence shows a fly-through of a CT scan of a human head, entering at the nose and moving further toward the pituitary gland.

These steps can all be performed in the same fragment program. Drawing the back faces of the bounding geometry results in the last intersection points of rays and active brick geometry, which are denoted as l_i in Figure 2.5. Subtracting end positions from start positions yields the ray vectors, which can then be normalized and stored in the RGB components of the *ray length image* together with the ray lengths in the alpha channel. Note that the alpha channel of the ray length image has consistently be set to zero where a ray should not be started at all, which is exploited in the ray casting pass.

2.3.2 Geometry Intersection

Many applications, e.g., virtual endoscopy, require both volumetric and polygonal data to be present in the same scene. Naturally, intersections of the volume and geometry have to achieve a correct visibility order, and in many cases looking at the intersections of the geometry and the isosurface is the reason for rendering geometry in the first place. Also, parts that do not contribute to the final image because they are occluded by geometry should not perform ray casting at all. An easy way to achieve this is to terminate rays once they hit a polygonal object by modifying the ray length image accordingly. This is illustrated in Figure 2.11. Of course, ray lengths should only be modified if a polygonal object is closer to the view point than the initial ray length. This problem can

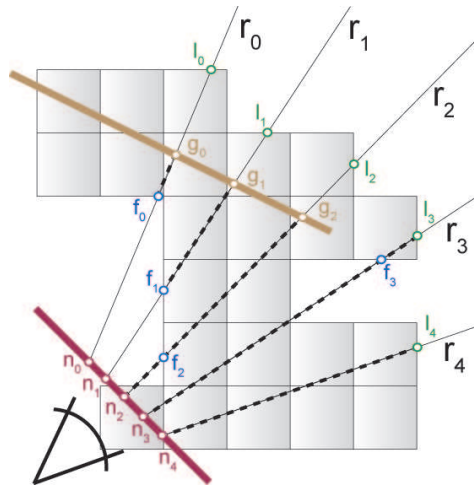


Figure 2.11: When rays intersect opaque polygonal geometry, they are terminated immediately. This is achieved by modifying the ray length image accordingly.

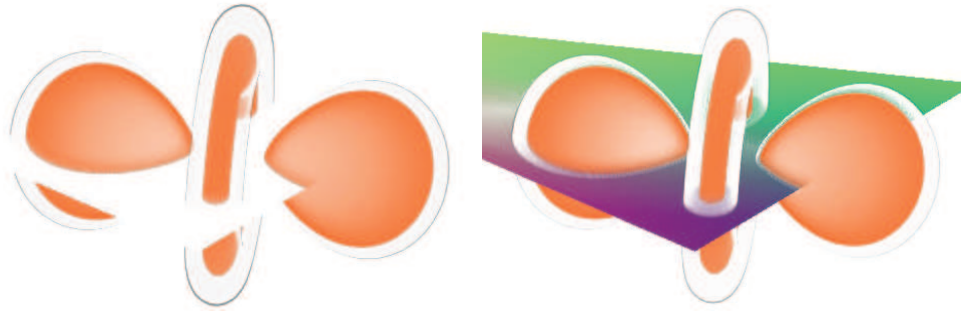


Figure 2.12: Modifying ray end positions prevents rendering occluded parts of the volume (left). Blending the result of ray casting on top of the opaque geometry then yields the correct result (right).

again be solved by using the depth test.

After rendering the back faces of active/inactive brick boundaries with their respective depth values (and depth test set to `GL_GREATER`), the intersecting geometry is rendered to the same buffer, with the corresponding volume coordinates encoded in the color channel. With the depth test reversed to `GL_LESS`, only those parts will be drawn that are closer to the view point than the initial ray lengths. This approach modifies ray casting such that it results in an image that looks as if it was intersected with an invisible object. Blending this image on top of the actual geometry in the last pass of the algorithm results in a rendering with correct intersections and visibility order.

2.4 Isosurface Ray Casting

This section describes a special case of volume ray casting for rendering isosurfaces, which is also known as *first-hit ray casting*. In order to facilitate object-order empty space skipping without per-sample overhead, we maintain min-max values of a regular subdivision of the volume into small blocks, e.g., with 4^3 or 8^3 voxels per block. These blocks do not actually re-arrange the volume. For each block, a min-max value is simply stored in an additional structure for culling. If the whole volume does not fit in GPU memory, however, a second level of coarser bricks can be maintained, which is described in later sections on memory management. Whenever the isovalue changes, blocks are culled against it using their

min-max information and a range query [12], which determines their active status. The view-independent geometry of active block bounding faces that are adjacent to inactive blocks is kept in GPU memory for fast rendering.

In order to obtain ray start depths $t_{start}(x, y)$, the front faces of the block bounding geometry are rendered with their corresponding distance to the camera. The front-most points of ray intersections are retained by enabling a corresponding depth test (e.g., `GL_LESS`). For obtaining ray exit depths $t_{exit}(x, y)$ we rasterize the back faces with an inverted depth test that keeps only the farthest points (e.g., `GL_GREATER`). Figure 2.6 shows that this approach does not exclude inactive blocks from the search range if they are enclosed by active blocks with respect to the current viewing direction. The corresponding samples are skipped on a per-sample basis early in the ray casting loop. However, most rays hit the isosurface soon after being started and are terminated quickly (yellow points in Figure 2.6, left). Only a small number of rays on the outer side of the isosurface silhouette are traced for a larger distance until they hit the exit position of the block bounding geometry (red points in Figure 2.6, left). The right side of Figure 2.6 illustrates the worst case scenario, where rays are started close to the view point, miss the corresponding part of the isosurface, and sample inactive blocks with image-order empty space skipping until they enter another part of the isosurface bounding geometry and are terminated or exit without any intersection. In order to minimize the performance impact when the distance from ray start to exit or termination is large, we use an adaptive strategy for adjusting the distance between successive samples along a ray.

2.4.1 Adaptive Sampling

In order to find the position of intersection for each ray, the scalar function is reconstructed at discrete sampling positions $\mathbf{x}_i(x, y) = \mathbf{c} + t_i \mathbf{d}(x, y)$ for increasing values of t_i in $[t_{start}, t_{exit}]$. The intersection is detected when the first sample lies behind the isosurface, e.g., when the sample value is smaller than the isovalue. Note that in general the exact intersection occurs somewhere between two successive samples. Due to this discrete sampling, it is possible that an intersection is missed entirely when the segment between two successive samples crosses the isosurface twice. This is mainly a problem for rays near the silhouette. Guaranteed intersections even for thin sheets are possible if the gradient length is bounded by some value L [48]. Note that for distance fields, L is equal

to 1. For some sample value f , it is known that the intersection at iso-value ρ cannot occur for any point closer than $h = |f - \rho|/L$. Yet, h can become arbitrarily small near the isosurface, which would lead to an infinite number of samples for guaranteed intersections.

We use adaptive sampling to improve intersection detection. The actual intersection position of an intersection that has been detected is then further refined using the approach described in Section 2.4.2. We

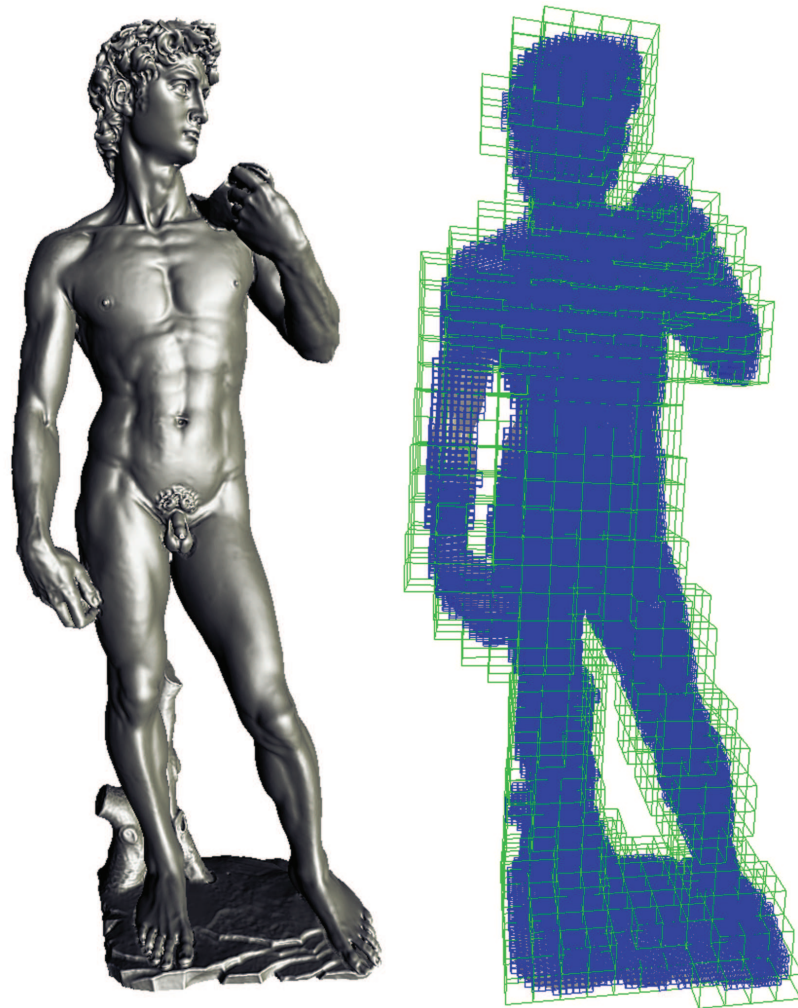


Figure 2.13: Michelangelo's David extracted and shaded with tri-cubic filtering as isosurface of a 576x352x1536 16-bit distance field [37]. The distance field is subdivided into two levels: a fine level for empty space skipping during ray casting (blue) and a coarse level for texture caching (green).

have found that completely adaptive sampling rates are not well suited for implementations on graphics hardware. These architectures use multiple pipelines where small tiles of neighboring pixels are scan-converted in parallel using the same texture cache. With completely adaptive sampling rate, the sampling positions of neighboring pixels diverge during parallel execution, leading to under-utilization of the cache. Therefore, we use only two different discrete sampling rates. The *base sampling rate* r_0 is specified directly by the user where 1.0 corresponds to a single voxel. It is the main tradeoff between speed and minimal sheet thickness with guaranteed intersections. In order to improve the quality of silhouettes (see Figure 2.14), we use a second *maximum sampling rate* r_1 as a constant multiple of r_0 : $r_1 = nr_0$. We are currently using $n = 8$ in our system. However, we are not detecting silhouettes explicitly at this stage, because it would be too costly. Instead, we automatically increase the sampling rate from r_0 to r_1 when the current sample's value is closer to the isovalue ρ by a small threshold δ . In our current implementation, δ is set by the user as a quality parameter, which is especially easy for distance fields where the gradient magnitude is 1.0 everywhere. In this case, a constant δ can be used for all data sets, whereas for CT scans it has to be set according to the data.

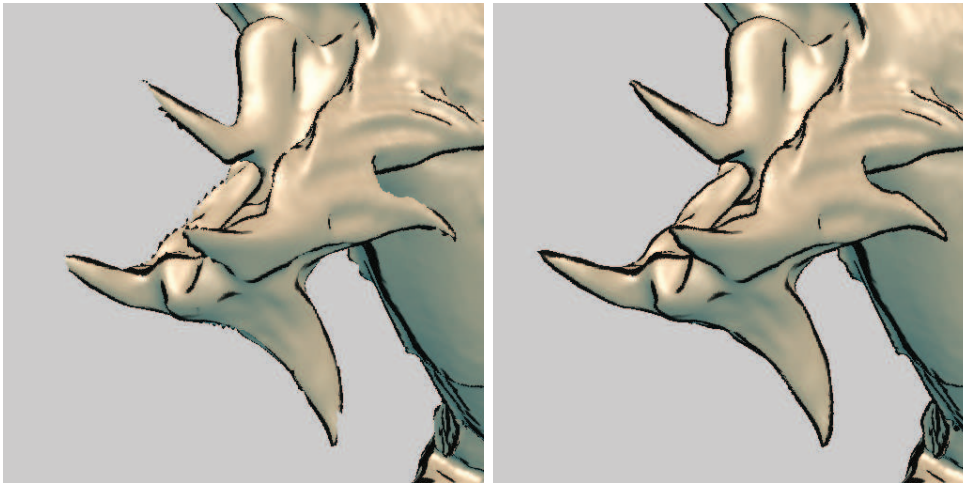


Figure 2.14: The left image illustrates a small detail of the asian dragon model with a sampling rate of 0.5. On the right, adaptive sampling increases the sampling rate to 4.0 close to the isosurface. Note that except at the silhouettes there is no visible difference due to iterative refinement of intersections.

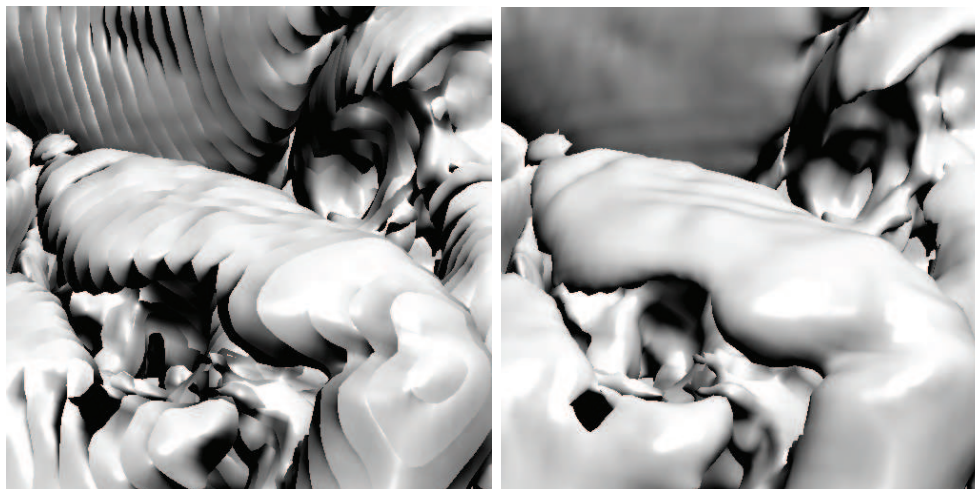


Figure 2.15: Enabling isosurface intersection refinement results in a huge improvement in image quality without any noticeable impact on performance.

2.4.2 Intersection Refinement

Once a ray segment containing an intersection has been detected, the next stage determines an accurate intersection position using an iterative bisection procedure. In one iteration, we first compute an approximate intersection position assuming a linear field within the segment. Given the sample values f at positions \mathbf{x} for the near and far ends of the segment, the new sample position is

$$\mathbf{x}_{new} = (\mathbf{x}_{far} - \mathbf{x}_{near}) \frac{\rho - f_{near}}{f_{far} - f_{near}} + \mathbf{x}_{near} \quad (2.1)$$

Then the value f_{new} is fetched at this point and compared to the isovalue ρ . Depending on the result, we update the ray segment with either the front or the back sub-segment. If the new point lies in front of the isosurface (e.g. $f_{new} > \rho$), we set \mathbf{x}_{near} to \mathbf{x}_{new} , otherwise we set \mathbf{x}_{far} to \mathbf{x}_{new} and repeat. Often a fixed number of iteration steps, e.g., four steps, is enough for obtaining high-quality intersection positions.

2.5 Memory Management

Volume sizes are increasing rapidly, and can easily exceed the available amount of GPU on-board memory. However, large parts of many types

of volumes are often mapped to optical properties such that they are completely transparent, e.g., the air around a medical or industrial CT scan. In order to decouple the amount of memory that is actually needed to render a given volume, i.e., the *working set* required for rendering it, from the overall volume size, a variety of memory management schemes such as bricking and, additionally, multi-resolution schemes, can be employed. We first consider the conceptually simple case of rendering iso-surfaces, which, however, almost directly extends to the case of direct volume rendering with arbitrary transfer functions.

For any possible isovalue, many of the blocks do not contain any part of the isosurface. In addition to improving rendering performance by skipping empty blocks, this fact can also be used for reducing the effective memory footprint of relevant parts of the volume significantly. Whenever the isovalue changes, the corresponding range query also determines the active status of bricks of coarser resolution, e.g., 32^3 voxels. These bricks re-arrange the volume and include neighbor samples to allow filtering without complicated look-ups at the boundaries, i.e., a brick of resolution n^3 is stored with size $(n+1)^3$ [54]. This overhead is inversely proportional to the brick size, which is the reason for using two levels of subdivision. Small blocks fit the isosurface tightly for empty space skipping and larger bricks avoid excessive storage overhead for memory management.

In order to decouple the volume size from restrictions imposed by GPUs on volume resolution (e.g., 512^3 on NVIDIA GeForce 6) and available video memory (e.g., 512MB), we can perform ray casting directly

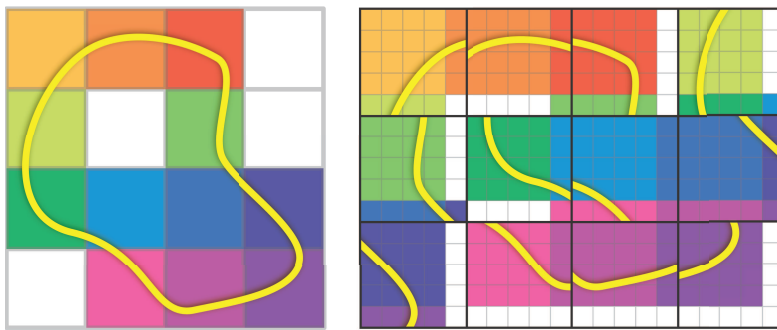


Figure 2.16: A low-resolution brick reference texture (left) stores references from volume coordinates to texture cache bricks (right). The reference texture is sampled in the fragment program to transform volume coordinates into brick cache texture coordinates. White bricks denote *null* references for bricks that are not resident in the cache.

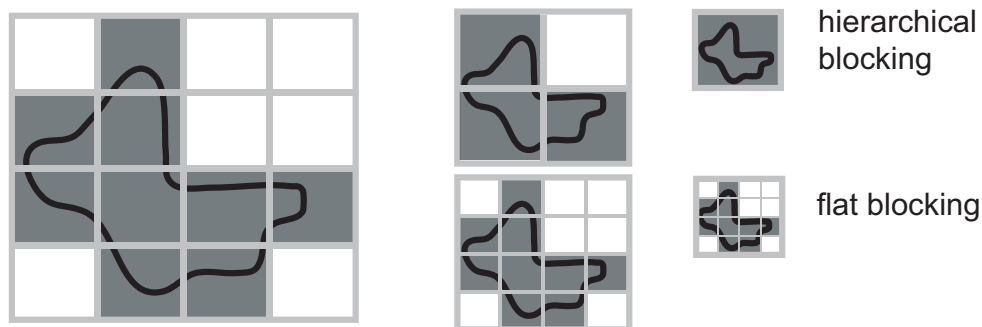


Figure 2.17: Hierarchical bricking (top row) vs. flat bricking (bottom row). Culled bricks are marked in white.

on a re-arranged brick structure. Similar to the idea of adaptive texture maps [54], we maintain an additional low-resolution floating point reference texture (e.g., 16^3 for a 512^3 volume with 32^3 bricks) storing texture coordinate offsets of bricks in a single brick cache texture that is always resident in GPU memory (e.g., a $512 \times 512 \times 256$ texture). However, both the reference and the brick cache texture are maintained dynamically and not generated in a pre-process [54]. Figure 2.16 illustrates the use of the reference and brick cache textures. Note that since no gradient reconstruction or shading is performed during ray casting, no complicated neighbor look-ups are required at this stage. When the isovalue changes, bricks that potentially contain a part of the isosurface are downloaded into the brick cache texture. Inactive bricks are removed with a simple LRU (least recently used) strategy when their storage space is required for active bricks. Bricks that are currently not resident in the cache texture are specially marked at the corresponding position in the reference texture (shown as white squares in Figure 2.16). During ray casting, samples in such bricks are simply skipped.

2.6 Mixed-Resolution Volume Rendering

Most multi-resolution volume rendering methods are based on hierarchical bricking schemes where the brick size in voxels is kept constant from level to level, and the spatial extent of bricks increases from high to low resolution until a single brick covers the entire volume (Figure 2.17, top row). Conversely, flat bricking schemes (Figure 2.17, bottom row) keep the spatial extent of bricks constant and successively decrease the brick

size in voxels. A major advantage of flat bricking schemes is that the culling rate is much higher, illustrated by the number of white bricks in Figure 2.17, because the granularity of culling stays constant irrespective of actual brick resolutions. This not only reduces the required texture memory, as more bricks can be culled, but also allows for a much more fine-grained LOD or fragment program selection per brick [61]. However, flat multi-resolution techniques have a bigger memory overhead when samples are replicated at brick boundaries, because for decreasing brick sizes the overhead of duplicated voxels increases. This overhead can be removed by avoiding sample duplication [65], trading off runtime filtering cost for memory savings. We employ flat multi-resolution bricking with sample duplication, but reduce the run-time overhead significantly by using hardware filtering and only warping the texture coordinates of samples where necessary [5].

2.6.1 Volume Subdivision for Texture Packing

The original volume is subdivided into equally-sized bricks of size n^3 in a pre-process, where n is a power of two, e.g., $n = 32$. During this subdivision, the minimum and maximum value in each brick are stored for culling later at run time, and lower-resolution versions of each brick are constructed. For the latter we compute the value of the new sample at the center of eight surrounding higher-resolution samples as their average, but higher-order filters could also be used. We limit the number of resolution levels to minimize the overhead of duplicated boundary voxels, and also to allow tight packing of low-resolution bricks in the storage space reserved for high-resolution bricks (Section 2.6.2). By default we use only two resolution levels, e.g., 32^3 bricks with a downsampled resolution of 16^3 . For fast texture filtering during rendering, voxels at brick boundaries are duplicated. In principle, duplication at one side suffices for this purpose [104], e.g., storing $(32+1)^3$ bricks. However, in the high-resolution level we duplicate at both sides, because the space for a single $(32+2)^3$ brick provides storage for eight $(16+1)^3$ bricks. Coincidentally, this often even does not impose additional memory overhead. The brick cache texture (Section 2.6.2) always has power-of-two dimensions for performance reasons, and a cache of size 512^3 , for example, can hold the same number of 34^3 and 33^3 bricks.

Although this approach is not fully scalable, it is very simple and a good trade-off that is not as restrictive as it might seem. Because culling is very efficient in a flat scheme, fewer bricks need to be resident in GPU memory. Even without culling, if the size of the brick cache

texture is $512 \times 512 \times 1024$ (256 mega voxels), for example, and two resolution levels are used (brick storage size 34^3), $15 \times 15 \times 30$ bricks fit into the cache. This yields a possible data set size of about 1.7 giga voxels, e.g., $960 \times 960 \times 1920$, if all bricks actually need to fit into the cache. Due to culling, the real data set size can typically be much larger. Additionally, for very large data three levels could be used. For example, increasing the allocated space for each brick from $(32 + 2)^3$ to $(32 + 4)^3$, both 16^3 and 8^3 bricks can be packed tightly, including boundary duplication for filtering. Using three levels with storage for $(32 + 4)^3$ bricks, $14 \times 14 \times 28$ bricks would fit into the cache, yielding a data set size of 10.7 giga voxels, e.g., $1792 \times 1792 \times 3584$, and more when bricks are culled.

2.6.2 Mixed-Resolution Texture Packing

For rendering, a list of active bricks is determined via culling, using, e.g., the transfer function or iso value, and clipping plane positions to determine non-transparent bricks that need to be resident in GPU memory. The goal is to pack all active bricks into a single 3D brick cache texture (Figure 2.18, right). In the beginning, all cache space is allocated for high-resolution bricks. If the number of active bricks after culling exceeds the allocated number, individual bricks are chosen to be represented at lower resolution. In this case, the effective number of bricks in the cache is increased by successively mapping high-resolution bricks in the cache to eight low-resolution bricks each, until the required overall number of bricks is available. This is possible because the storage allocation for bricks has been chosen in such a way that exactly eight low-resolution bricks fit into the storage space of a single high-resolution brick, including duplication of boundary voxels, as described in the previous section.

After the list of active bricks along with the corresponding resolutions has been computed, the layout of the cache texture and mapping of brick storage space in the cache to actual volume bricks can be updated accordingly, which results in an essentially arbitrary mixture of resolution levels in the cache. The actual brick data are then downloaded into their corresponding locations using, e.g., `glTexSubImage3D()`. During rendering, a small 3D layout texture is used for address translation between “virtual” volume space and “physical” cache texture coordinates (Figure 2.18, top left), which is described in the next section.

2.6.3 Address Translation

A major advantage of the texture packing scheme described here is that address translation can be done in an identical manner irrespective of whether different resolution levels are mixed. Each brick in virtual volume space always has constant spatial extent and maps to exactly one brick in physical cache space. “Virtual” addresses in volume space, in $[0, 1]$, corresponding to the volume’s bounding box, are translated to “physical” texture coordinates in the brick cache texture, also in $[0, 1]$, corresponding to the full cache texture size, via a lookup in a small 3D layout texture with one texel per brick in the volume. This layout texture encodes (x, y, z) address translation information in the RGB color channels, and a multi-resolution scale value in the A channel, respectively. A volume space coordinate $\mathbf{x}_{x,y,z} \in [0, 1]^3$ is translated to cache texture coordinates $\mathbf{x}'_{x,y,z} \in [0, 1]^3$ in the fragment program as:

$$\mathbf{x}'_{x,y,z} = \mathbf{x}_{x,y,z} \cdot \mathbf{bscale}_{x,y,z} \cdot \mathbf{t}_w + \mathbf{t}_{x,y,z}, \quad (2.2)$$

where $\mathbf{t}_{x,y,z,w}$ is the RGBA-tuple from the layout texture corresponding to volume coordinate $\mathbf{x}_{x,y,z}$, and \mathbf{bscale} is a constant fragment program parameter containing a global scale factor for matching the different coordinate spaces of the volume and the cache. When filling the layout

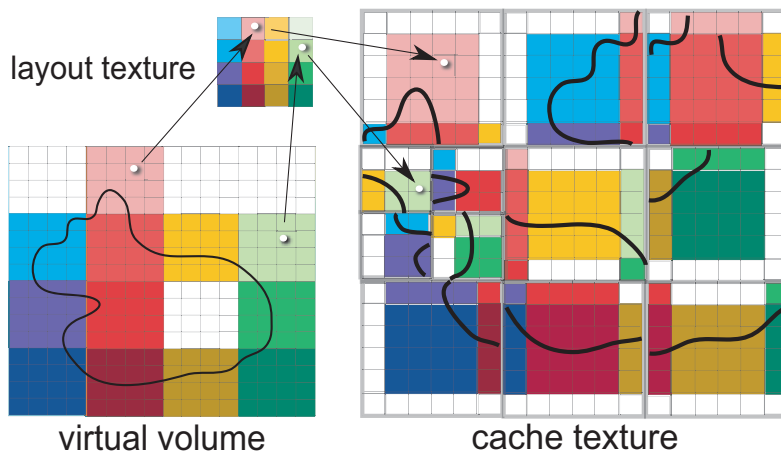


Figure 2.18: Mixed-resolution texture packing and address translation from virtual volume space to physical cache texture space via the layout texture. Resolution levels are mixed by packing low-res bricks tightly into high-res bricks.

texture, the former is computed as:

$$\mathbf{t}_{x,y,z} = (\mathbf{b}'_{x,y,z} \cdot \mathbf{bres}'_{x,y,z} - \mathbf{o}_{x,y,z} + \mathbf{t}_w) / \mathbf{csize}_{x,y,z} \quad (2.3)$$

$$\mathbf{t}_w = 1.0, \quad (2.4)$$

for a high-resolution brick, where \mathbf{b}' is the position of the brick in the cache (0, 1, ...), \mathbf{bres}' is the storage resolution of the brick, e.g., 34^3 , and \mathbf{csize} is the cache texture size in texels to produce texture coordinates in the $[0, 1]$ range. For a low-resolution brick, this is computed with $\mathbf{t}_w = 0.5$. The offset $\mathbf{o}_{x,y,z}$ is computed as:

$$\mathbf{o}_{x,y,z} = \mathbf{b}_{x,y,z} \cdot \mathbf{bres}_{x,y,z} \cdot \mathbf{t}_w, \quad (2.5)$$

where \mathbf{b} is the position of the brick in the volume (0, 1, ...), and \mathbf{bres} is the brick resolution in the volume, e.g., 32^3 . The global scale factor \mathbf{bscale} is computed as:

$$\mathbf{bscale}_{x,y,z} = \mathbf{vsize}_{x,y,z} / \mathbf{csize}_{x,y,z}, \quad (2.6)$$

where \mathbf{vsize} is the size of the volume in voxels.

2.7 Multiresolution LOD and Adaptive sampling

2.7.1 Octree-based Multiresolution Representation

The linear storage scheme described above has a poor data locality property. The lookup of neighboring voxels is frequent and it is only along the x -axis that this translates to access of neighboring memory locations. The impact of cache-misses in rendering and processing is significant and often causes a scheme to ultimately fail if not well addressed. Blocking of the volume is therefore generally efficient and significantly improves the cache hit-rate.

The size of a block is typically derived from the size of the level 1 and 2 caches. Grimm et al. [32, 31] finds that a block size of 32, $\mathbf{B} = (32, 32, 32)$, is the most efficient for their block-based raycaster. Parker et al. [78] use a smaller block size, $\mathbf{B} = (4, 4, 4)$, for a parallel iso-surface¹ renderer. This block size matches the size of the L1 cache line on SGI super-computers (SGI Onyx2 & SGI Origin 2000). In numerous publications it is indicated that blocking by 16 or 32 is an optimal size for many block related processing tasks.²

The addressing of blocks and samples within blocks is straightforward, but introducing a block map structure allows for arbitrary placement of blocks and packing in memory with unused blocks being ignored and thus saving memory space. The introduction of blocking results in an additional level of complexity for block boundary handling, especially for the cases when a sample is requested in a neighboring block that has been ignored. Two strategies can be explored to deal with this. The first requires the access of neighboring blocks. Grimm et al. [32], for example, propose a scheme based on table lookups for neighboring samples that avoids conditional branches in the code. The second strategy is based on self-contained blocks and requires the replication of neighboring samples. The overhead for sample replication is less than 20% for block sizes of 16 and up.

Additional basic data conversions may also be applied, such as remapping the value range and conversion to 8- or 16-bit integers, that is data types that directly map to native GPU types. Blocking improves memory locality for software-based rendering and processing. Skipping empty

¹An iso-surface, S , is an implicit surface defined as $S = \{ \mathbf{p} \mid s(\mathbf{p}) = C \}$, where C is a constant.

²In two-dimensional blocking, or tiling, the equivalent size is 64×64 , also being the default tile size in JPEG-2000 [1].

blocks usually has a significant effect on the data size and rendering performance. The concept of an empty block, however, needs to be clarified and defined, which is an integral part of the next section.

2.7.2 Block Properties and Acceleration Structures

In order to reveal any embedded entities within a volume it is obvious that some samples must be rendered transparent and other samples rendered semi-transparent or opaque. This is achieved through the use of a Transfer Function (TF). For a blocking scheme, as described above, the meaning of an *empty* block is a block that has all its voxels classified as completely transparent. Naturally, such a block could be discarded in the rendering process and thus improve the performance. Since the goal is to reduce the amount of data in the pipeline it is essential that empty blocks can be predicted without access to all samples in a block. Meta-data for such predictions is collected during preprocessing, and preferably without knowledge of specific TF settings.

The TF usually defines one or more regions in the scalar range as non-transparent and, for the rendering of iso-surfaces, either narrow peaks are defined or special iso-surface renderers are used. It is therefore natural that ideas from iso-surface extraction acceleration schemes have been applied. The goal of these schemes are to minimize the processing so that only cells intersecting the iso-surface are considered. Wilhelms & Gelder [108] create a tree of min/max values. The tree is created bottom up and starts with the cells, cubes of 8 voxels. Livnat et al. [62] extend this approach and introduce the span-space. For iso-surface rendering, a leaf in the tree is included if the iso-value is within the range spanned by the minimum and maximum value of the cell. Parker et al. [78] use a limited two-level tree and find that sufficient in their software implementation of an iso-surface raycaster.

For arbitrary TF settings, the min/max scheme is generally overly conservative and may classify empty blocks as non-empty. Summed-Area Tables [17] of the TF opacity are used by Scharsach [91] to determine the blocks' content by taking the difference of the table entries for the minimum and maximum block values. The low granularity of the min/-max approach is addressed by Grimm et al. [31] who, instead, use a binary vector to identify block content. The scalar range is quantized into 32 uniform regions and the bit-vector indicates the presence of samples within the corresponding range. A similar approach is taken by Gao et al. [27] but they use a larger vector, matching the size of their TF table (256 entries).

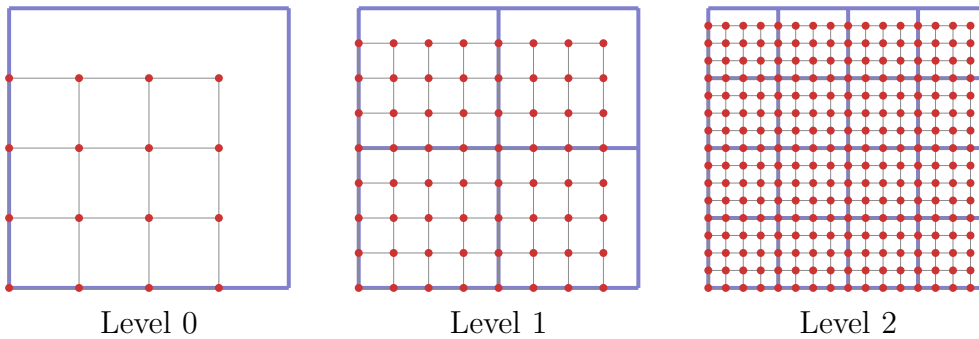


Figure 2.19: Hierarchical blocking with subsampling. Downsampling is achieved by removing every even sample [56] or by a symmetric odd-sized filter [105].

2.7.3 Hierarchical Multiresolution Representations

Simply skipping empty blocks might not reduce the volume size sufficiently, the total size of the remaining non-empty blocks may still be above the available memory size. A strategy is then to apply techniques that vary the resolution in different parts of the volume, so different blocks in the volume have different resolutions. This Level-of-Detail (LOD) approach enables a more graceful adaptation to limited memory and processing resources.

The most common scheme is to create a hierarchical representation of the volume by recursive downsampling of the original volume. Since each lower resolution level is $1/8$ the size of the previous, the additional amount of memory required for this pyramid is less than 14.3%. The created hierarchies may differ depending on the selected downsampling scheme. Figure 2.19 illustrates three levels of an hierarchy created using subsampling, every second sample being removed. This scheme is used by LaMar et al. [56] and Boada et al. [8], amongst others. Weiler et al. [105] also use this placement of samples but employ a quadratic spline kernel in the downsampling filter since they argue that subsampling is a poor approximation.

The positions of the downsampled values, however, do require some attention. The positioning indicated in figure 2.19 skews the represented domain. A more appropriate placing of a downsampled value is in the center of the higher resolution values it represents. This placement is illustrated in figure 2.20 and is also a placement supported by average downsampling.

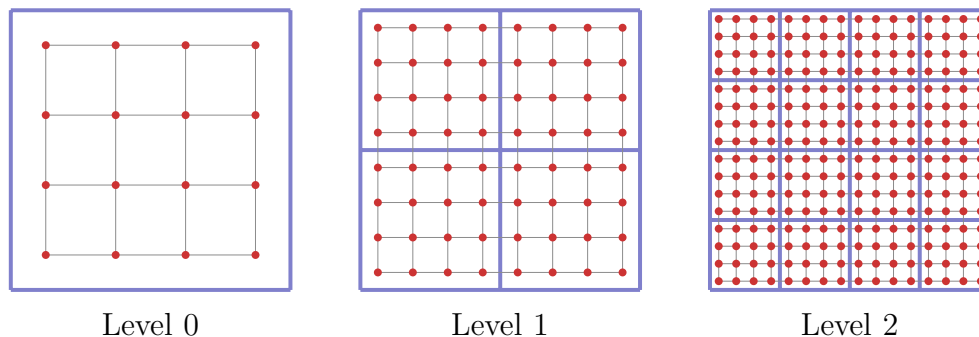


Figure 2.20: Hierarchical blocking with average downsampling.

In order to be able to select different resolution levels in different parts of the volume blocking is suitable for hierarchical representations as well. The block size, in terms of number of samples, is usually kept equal at each resolution level and the block grids are indicated by wide, blue lines in the figures 2.19 and 2.20. Blocks at lower resolutions cover increasingly large spatial extents of the volume. These multiresolution hierarchies thus provide supporting data structures for LOD selection. Methods to determine an appropriate level of detail are discussed in the following section.

2.8 Level-of-Detail Management

It is not sufficient to only determine if a block is empty or not. The multiresolution representations described above require additional and different techniques that also can determine resolution levels for the blocks. This section reviews techniques and approaches for LOD selection that have been suggested in the literature. These approaches can be classified into: view dependent and region-of-interest, data error, and transfer function based techniques. It is, furthermore, common to combine several of these measures in different configurations. The following sections will, however, review them individually.

The conceptual principle for hierarchical LOD selection is similar for all approaches. The selection starts by evaluating one or more measures for a root node. If the resolution of a block, a node in the hierarchy, is found adequate then the traversal stops and the selection process is done. If the resolution needs to be increased the block is either immediately replaced by all its children or a subset of the children is added. The

latter approach will remove the parent node when all its children have been added. If the amount of data to use is limited, this constraint is checked at every step and the LOD selection is stopped when the limit is reached.

2.8.1 View-Dependent Approaches

View-dependent techniques seek to determine the LOD selection based on measures like distance to viewer and projected screen-space size of voxels. Region-of-interest methods work similarly to distance to viewer measures. Using full resolution blocks when viewing entire volumes can be suboptimal. When a single pixel covers multiple voxels it may result in aliasing artefacts. Reducing the resolution of the underlying sampled data (prefiltering) is, in fact, standard in graphics rendering instead of supersampling. It is referred to as mipmapping³ in the graphics literature.

Distance to viewer approaches are used in [56, 105, 35, 6], for instance. A block is refined if the projected voxel size is larger than one pixel on the screen, for example. The distance to viewer or region-of-interest can furthermore be used to weight some other measure, like a data error measure, by dividing that measure by the distance.

2.8.2 Data Error Based Approaches

Representing a block in a volume with a lower resolution version may naturally introduce errors when the volume is sampled compared with using the full resolution. A measure of this error, for instance the Root-Mean-Square-Error (RMSE), expresses the amount of error introduced. When selecting a LOD for the multiresolution hierarchy, the block with the highest data error should be replaced with a higher resolution version. Repeating this procedure until the memory budget is reached will then select a level-of-detail for the volume that minimizes the data error. This measure only depends on the data and can therefore be computed in the preprocessing step.

This approach is used in Boada et al. [8], who also take into account the effect of linear interpolation in the lower resolution version. In addition, a user-defined minimum error threshold is used to certify that the represented data correspond to a certain quality. Guthe et al. [35]

³Mip is an abbreviation of the Latin *multum in parvo* – many things in a small place.

also take this approach, using the L_2 -norm, but combine it with view-dependent measures, namely distance-to-viewer and projected voxel size.

2.8.3 Transfer Function Based Approaches

The shortcoming of data error approaches lies in the mapping of data samples through the TF. The content of the TF is arbitrary and consequently the data error is a poor measure if it is used for volume rendering. Determining the content of a block in the TF domain has a higher relevance since this will affect the quality of the rendered image more directly. The notion of a block's *TF content* is explored below and several schemes for TF content prediction are reviewed. The challenge, however, is to predict the required LOD for each block without accessing the data beforehand.

The complete distribution of sample values within a block is a highly accurate description of the block content, losing only spatial distribution. Such a description could, however, easily result in meta-data of significant sizes, potentially larger than the block data itself. LaMar et al. [57] therefore introduce frequency tables to express the frequency of specific data errors (differences) and compute an intensity error for a greyscale TF as an approximation to the current TF. Guthe et al. [34] instead use a more compact representation of the maximum deviation in a small number of bins, for which the maximum error in RGB-channels are computed separately. A combined approach of these two, using smaller binned frequency tables, is presented by Gyulassy et al. [36].

Gao et al. [27] use a bit-vector to represent the presence of values in a block. The block vector is gated against RGB bit-vectors of the TF. If the difference of two such products, compared with a lower resolution block level, is less than a user defined threshold then the lower resolution block can be chosen instead. A similar approach using a quantized binary histogram is presented in [31] but is not reported to be used for LOD selection.

2.9 Encoding, Decoding and Storage

In section 2.7.3 a conceptual view of multiresolution hierarchies was described. As mentioned, the amount of data is not reduced by this process, rather increased. When the amount of data in the hierarchy can not be handled in core memory, additional techniques are required. Data compression is one viable approach and, specifically, lossy compression can

significantly reduce the amount of data, at the cost of a loss of fidelity. Another approach is to rely on out-of-core storage of the volume hierarchy and selectively load requested portions of the data. A combination of these techniques is also possible. Some of the well-known approaches are described in the following sections.

2.9.1 Transform and Compression Based Techniques

Following the success of image compression techniques, it is natural that such techniques be transferred to volumetric data sets. Usually a transform is applied to the data and it is the coefficients from the transform that are stored. The underlying idea for the transform is to make data compression more efficient. Applying a compression technique on the coefficients, such as entropy coding, then yields a higher degree of compression compared to compressing the original data. The following sections review two transforms that are common for image compression and have been used for volume data. Basic concepts of compression are also presented.

2.9.1.1 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) is well established in image and video coding standards, such as JPEG and MPEG, and there exist highly optimized algorithms and code to perform this transform, with a typical block size of 8. Relatively few researchers have applied this transform to volumetric data sets although some examples exist [112, 79, 69]. Lum et al. [70] apply the DCT for time-resolved data. Instead of computing the inverse DCT, it is replaced by a texture lookup since the DCT coefficients are dynamically quantized and packed into a single byte.

2.9.1.2 Multiresolution Analysis – Wavelets

The wavelet transform has gained a wide acceptance and has been embraced in many application domains, specifically in the JPEG-2000 standard, described by Adams [1]. A significant amount of work on volume data compression has employed wavelet transforms. Being a multiresolution analysis framework it is well suited for the multiresolution handling of volume data. Several wavelets exist, but the Haar and LeGall (a.k.a. 5/3) integer transforms, supporting lossless encoding [9, 100], and the

Daubechies 9/7 [19] are the most common and can be efficiently implemented using the lifting scheme [99].

Conceptually, the transform applied on a 1D signal produces two sub-band signals as output, one describing low frequency content and the other describing high frequency content. Recursive application of the transform on the low frequency output produces multiple sub-band descriptions until a lowest resolution level is reached. Once the multiband analysis is done, the inverse transform can be applied in an arbitrary number of steps until the original full resolution has been reconstructed. For every step, the resolution of the volume is increased, doubled along each dimension. It is furthermore possible to apply the inverse transform selectively and retrieve higher resolution in selected subregions of the volume. This approach is taken by Ihm & Park [45], Nguyen & Saupe [76] and Bajaj et al. [2]. Their work is primarily concerned with efficient random access of compressed wavelet data and caching of fully reconstructed parts.

A wavelet transform can also be applied on blocks individually. Guthe et al. [35] collect eight adjacent blocks and apply the transform once on the combined block. The low frequency sub-band then represents a downsampled version. The high frequency sub-band, the detail, is compressed and stored separately. The procedure is then repeated recursively on each level until a single block remains. A hierarchy of detail data, high frequency coefficients, is thus constructed and can be selectively used to reconstruct a multiresolution level-of-detail selection. A hardware supported application of this technique, using a dedicated FPGA-board, is presented by Wetekam et al. [107].

2.9.1.3 Data Compression

Applying transforms to volume data does not reduce the amount of data. Instead, it frequently increases the size of the data. The Haar and LeGall integer wavelet bases, for example, require an additional bit per dimension and level. Nevertheless, as the entropy of the transformed signal is generally reduced, compared with the original signal, a compression scheme yields a higher compression ratio for the transformed signal. Lossless compression is, however, quite limited in its ability to reduce the data size. In many practical situations with noisy data, ratios above 3:1 are rare. Even this small reduction can be valuable but should be considered against the increased computational demand of decoding the compressed data stream and applying the inverse transform.

Significant data reduction can be achieved, however, if lossy com-

pression is allowed. Quantization of the coefficients is commonly used and can be combined with thresholding to remove small coefficients. Hopefully this results in many long sequences of zero values that can be compactly represented using run-length encoding. There exist a wide range of quantization and compression methods presented in the literature and several of these are used in the context of volume data compression [106, 45, 2, 33, 70]. Vector quantization of coefficient vectors is also applied [92, 69]. For reasonable distortion, causing minor visual degradation, compression ratios of 30:1 are achieved [35].

It is also reasonable to allow the encoding stage to take a significant processing time if the results are improved, it is the performance of the decoding stage that is critical.

2.9.2 Out-of-Core Data Management Techniques

Computer systems already employ multiple memory level systems, commonly two levels of cache are employed to buffer data from main memory. Since the cache memory is faster, this helps to reduce data access latency for portions of the data already in the cache. Extending caching techniques to include an additional layer, in the form of disk storage or on a network resource, is therefore quite natural and beneficial. A data set can be significantly larger than core memory but those subsets of the data to which frequent access is made can be held in core making these accesses much faster. Indeed, several techniques exist in operating systems that exploit this concept, for instance memory-mapped files.

The semantic difference between general purpose demand-paging is that an application may know significantly more about data access patterns than a general low level scheme could. Cox & Ellsworth [15] present application controlled demand-paging techniques and compare those with general operating system mechanisms, showing significant improvements for application-controlled management. Their work is applied to Computational Fluid Dynamics (CFD) data. Another example is the data querying techniques for iso-surface extraction presented by Chiang et al. [13]. The OpenGL Volumizer toolkit, presented by Bhaniramka & Demange [6], also supports management of large volumes and volume roaming on SGI graphics systems. Volume roaming provides scanning through the volume, with a high-resolution region of interest, and large blocks, $\mathbf{B} = (64, 64, 64)$, are loaded from high-performance disk systems on demand.

Distributed rendering approaches also make use of out-of-core data management ideas, the capacity of each rendering node is limited and

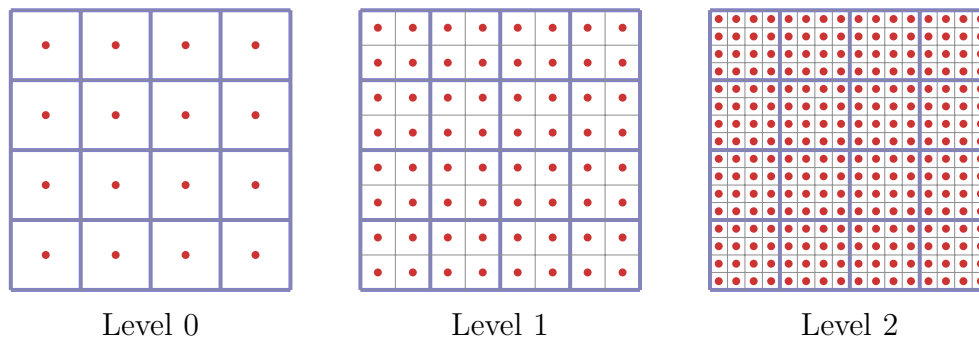


Figure 2.21: Flat multiresolution blocking. Spatial position and size is constant for all blocks (blue squares). The resolution of each block is arbitrary and independent of the resolution level of neighboring blocks.

data transport is costly. Predictive measures are required to ensure that the rendering load is balanced between the nodes. This is another aspect where application controlled data management is preferred, since the access latency can be reduced or hidden. Examples of this approach are presented in several papers by Gao et al. [28, 29, 27].

2.9.3 Flat blocking Multiresolution Representation

In flat blocking the samples on the uniform grid are centered on the grid cells instead of on the cell vertices. This sample placement is shown in figure 2.21 (grid in black and samples in red). This cell-centered sample placement is compatible with OpenGLs sampling location for the pixels in a framebuffer and texels in the textures. Block data is also cell centered on the block grid, as indicated by the blue block grid.

Furthermore, a multiresolution representation is created individually for each block, either by a wavelet transform [66] or by average down-sampling [65, 63, 67]. This scheme is referred to as a *flat* multiresolution blocking, or flat blocking, since no global hierarchy is created. The spatial extent of a block is constant and the blocks' spatial extents do not grow with reduced resolution level. The key advantages of the flat scheme can be summarized by:

- A uniform addressing scheme is supported.
- The granularity of the level-of-detail selection remains fine-grained.

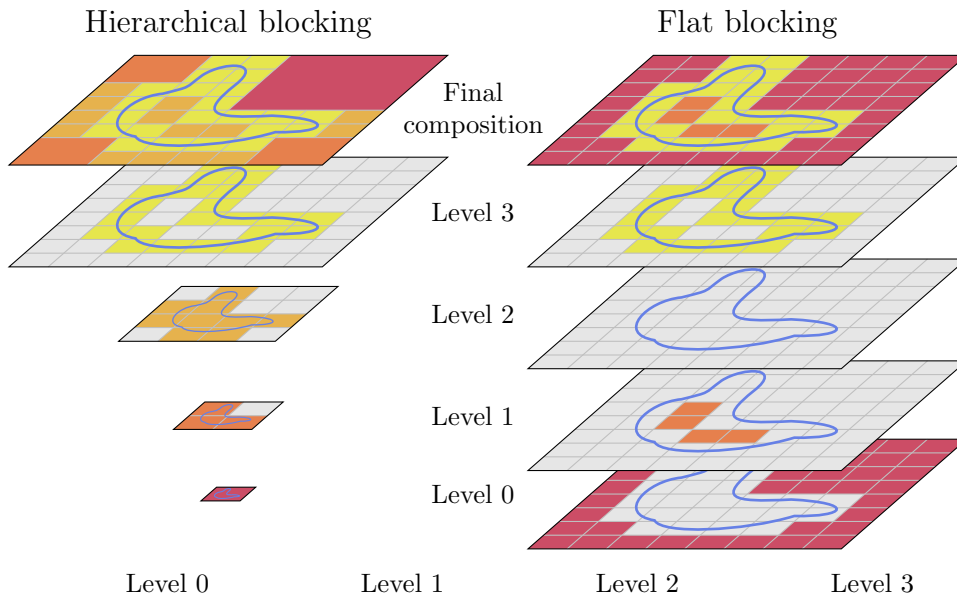


Figure 2.22: Comparing hierarchical and flat multiresolution blocking. Level 0 is the lowest resolution level and level 3 is the highest one. Levels are selected such that if a block intersects the boundary of an object (blue), that block is selected at the highest resolution and the interior (homogeneous) is selected at level 1.

- Arbitrary resolution differences between neighboring blocks can be supported since a block is independent of its neighbors.
- The resolution of a block is not restricted to be in powers-of-two.
- A heuristic analysis shows that flat blocking provides a higher memory efficiency than a corresponding cut through a hierarchical scheme, see table 2.1.

The disadvantage with this fine-grained flat blocking is that the number of blocks is constant. Hierarchical schemes scale in this respect with reduced memory budget. On the other hand, since there are no hierarchical dependencies it is trivial to exploit parallelism in many processing tasks for flat multiresolution data.

Figure 2.22 shows an illustrative comparison between hierarchical and flat multiresolution blocking. In this example the LOD is selected so that a block that intersects the boundary of the embedded object must have full resolution, while the blocks on the interior should be at the second lowest resolution, level 1. Blocks on the exterior are to be ignored, level 0. The LOD selection is indicated with level specific

Table 2.1: Memory efficiency for multiresolution volumes. The LOD selection criteria used in figure 2.22 are applied. Hierarchy 1 corresponds to a LOD selection where all children always replace a parent. Hierarchy 2 is more flexible and allows partial usage, as shown in figure 2.22.

Level	Hierarchy 1		Hierarchy 2		Flat	
	Blocks	Bytes	Blocks	Bytes	Blocks	Bytes
Level 0	0	0	1	64	38	38
Level 1	1	64	3	192	4	16
Level 2	3	192	9	576	0	0
Level 3	36	2304	22	1408	22	1408
Total	40	2560	35	2240	64	1462
Reduction	1.60:1		1.83:1		2.80:1	

coloring. Two strategies for block refinement are used in the literature, as discussed in section 2.7.3. The first is to completely replace a block with all its children. The second allows partial usage of a block with only the required higher resolution blocks being added and a part of the lower resolution block can still be used. These are referred to as Hierarchy 1 and 2 in table 2.1 which shows memory efficiency for both hierarchical LOD selection schemes and the flat scheme. Full resolution blocks have $8 \times 8 = 64$ samples and the original image size is 64×64 . The achieved data reduction using flat blocking is 2.8:1, compared to 1.6:1 and 1.8:1 for the two hierarchical LOD selection approaches.

2.10 Sampling of Multiresolution Volumes

The flat multiresolution structure provides a uniform addressing scheme for access to blocks and samples. The volume range is defined by the number of blocks, N_x, N_y, N_z , along each dimension. The block index, \mathbf{E} , is then easily retrieved as the integer part of a position, \mathbf{p} , within the volume. The remainder of the position then defines the intrablock local coordinate, $\mathbf{p}' = \text{frac}(\mathbf{p})$. The block index map, holding the block size, σ , and the location, \mathbf{q} , of the block in the packed volume, is then used to compute the coordinate for the sample to take. The coordinate range for flat block addressing is defined in units of blocks. Taking the integer part of a coordinate then yields the block index and the remainder yields the local, intrablock coordinate. This scheme is in effect virtualizing the volume address domain. The block index is used to lookup a block's

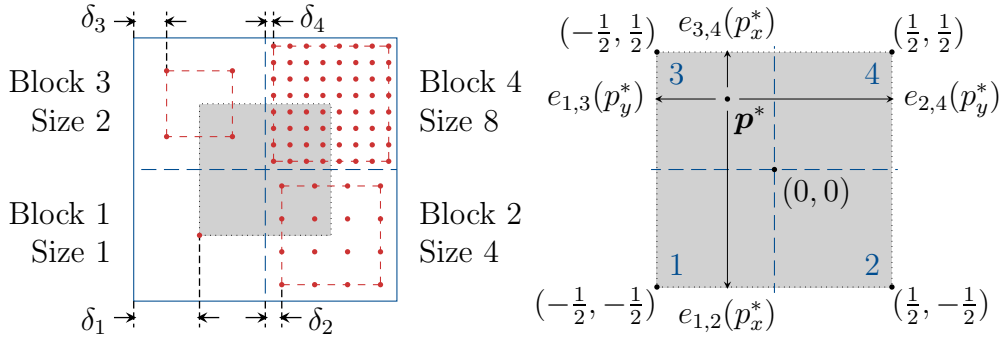


Figure 2.23: Illustration of block neighbors in 2D. The blocks are assigned different resolutions and the samples are indicated by the red dots. The distance between the sample boundaries (dashed lines in red) and the blocks' spatial boundaries (blue lines) are denoted by δ_i , for each block, i . The shaded area between block centers indicates the domain for local interblock coordinate, \mathbf{p}^* . Edge weights, $e_{i,j}$, are defined along the edges between adjacent blocks.

scale and location in the packed volume. Since blocks in the packed volume are rarely neighbors in the spatial domain, special care has to be taken in the sampling of a block. Furthermore, the scale of a block is arbitrary which also has implications for block sampling. An illustration of a neighborhood of four blocks is shown in figure 2.23.

2.10.1 Nearest Block Sampling

The first approach to block sampling is to restrict the sampling to access only data within the current block, suitably expressed in terms of operations on the intrablock coordinate, \mathbf{p}' . The valid coordinate domain for intrablock samples is indicated by squares of red, dashed lines in figure 2.23. The inset from the blocks' spatial boundaries is indicated by δ_i , for each block, i . The restricted sample location, \mathbf{p}'_C , is then defined as

$$\mathbf{p}'_C = \mathbf{C}_{1-\delta}^\delta(\mathbf{p}'), \quad (2.7)$$

where $\mathbf{C}_\alpha^\beta(x)$ clamps the value, or element values for vectors, of x to the interval $[\alpha, \beta]$. This form of sampling has been relabeled Nearest Block (NB) sampling in paper [67], but was introduced as *no interblock interpolation* in paper [65]. The GPU-based raycaster introduced in paper [63] also used NB sampling only. It is evident that block artefacts will arise, but these are not always visible and it is noted below that they are most apparent for thin, iso-surface like TF settings.

2.10.2 Interblock Interpolation Sampling

To overcome these block artefacts, an interblock interpolation technique was developed and introduced in paper [65]. Previous techniques rely on sample replication and padding between the blocks [56, 105, 35]. Replication, however, counteracts the data reduction in the pipeline and may also distort samples, where a block has higher resolution than its neighbor, in order to reduce interpolation discontinuities. Specifically for lower resolution blocks, and implicitly for higher data reduction, the data overhead becomes increasingly large, as shown in figure [65]:1b.

Interblock Interpolation (II) removes the need for sample replication and is a scheme for direct interpolation between blocks of arbitrary resolution, including non-power-of-two sizes. The principle for II sampling is to take a sample from each of the immediate closest neighboring blocks using NB sampling and compute a sample value by a normalized weighted sum. The domain for interblock interpolation in a neighborhood is indicated by the shaded area between the block centers in figure 2.23. The interblock local coordinate, $\mathbf{p}^* = \text{frac}(\mathbf{p} + 0.5) - 0.5$, has its origin at the intersection of the adjacent blocks.

The block weight, ω_b , for each of the blocks is computed using individual edge weights, $e_{i,j}$, for the edges between two facing block neighbors centers, blocks i and j , as illustrated in 2D in figure 2.23 by the grey dotted box edges between block centers. In figure 2.24 four different edge weight schemes are shown, including the NB sampling mode. The Maximum Distance scheme provides the smoothest interpolation, but is also the most sensitive to poor choice of level-of-detail since the filter kernel has the widest weighted support. Consider, for instance, the case where an iso-surface intersects a block outside its sample boundary. If a neighboring block is chosen at a low level-of-detail, the surface would bend out towards the low resolution block. Therefore the block value distributions used for LOD selection includes one layer of samples around each block.

The 2D version of the II sample scheme is then succinctly defined by the following normalized sum:

$$\varphi = \frac{\sum_{b=1}^4 \omega_b \varphi_b}{\sum_{b=1}^4 \omega_b}, \quad (2.8)$$

where φ_b is an NB sample from block b and the block weights, ω_b , are

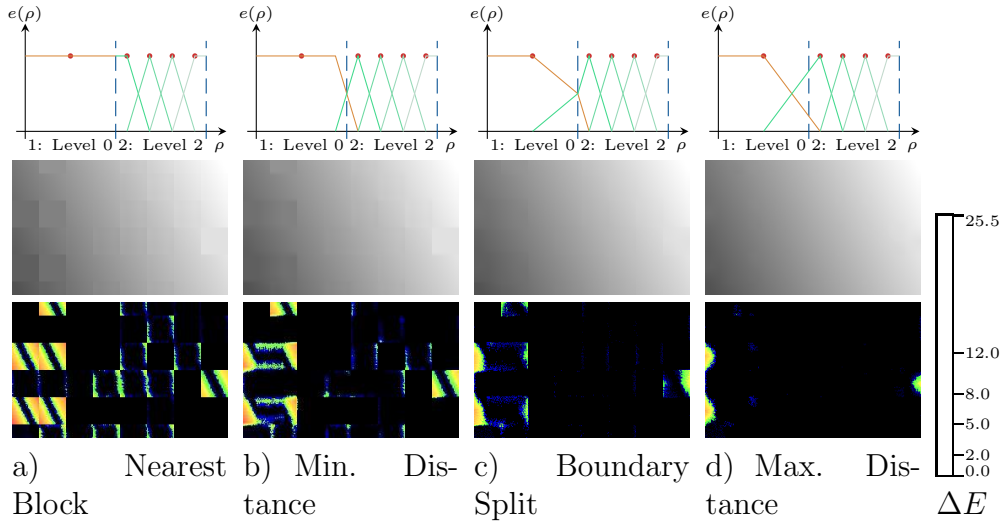


Figure 2.24: A comparison of the interpolation methods on a slightly rotated linear gradient. The original image was a 256×256 grayscale image (8×8 blocks) being reconstructed using random levels between 1×1 and the full resolution, 32×32 . The bottom row shows color mapped images of the pixel-wise errors using the CIE $L^*u^*v^*$ ΔE color difference. The images have been cropped.

defined as

$$\begin{aligned}\omega_1 &= (1 - e_{1,2}) \cdot (1 - e_{1,3}), \\ \omega_2 &= e_{1,2} \cdot (1 - e_{2,4}), \\ \omega_3 &= (1 - e_{3,4}) \cdot e_{1,3}, \\ \omega_4 &= e_{3,4} \cdot e_{2,4}.\end{aligned}$$

The corresponding 3D variant and definitions of the edge weight functions are presented in paper [65]. All edge weight functions described therein result in interpolations equivalent to trilinear interpolation whenever the neighboring blocks are all of equal resolution. It is also shown that the method constitutes a \mathcal{C}^0 continuous function.

2.10.2.1 Sample Placement Discussion

The placement of sample points within a block depends on the alignment of the superimposed block grid and on the scheme being used to derive lower resolution blocks. The placement used in figure 2.23 can be motivated for average value downsampling and some wavelet transforms, such as the Haar and the LeGall 5/3 wavelets [9, 100]. Other approaches for lower resolution representations might suggest different placements. Sample replication techniques use a different sample placement scheme

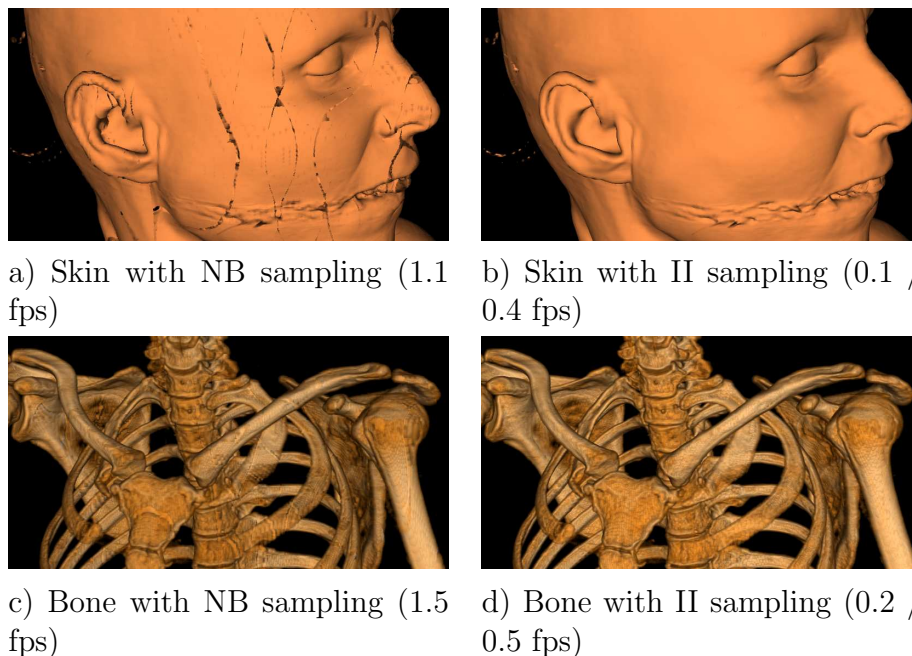


Figure 2.25: Comparison between Nearest Block and Interblock Interpolation sampling. Images are rendered with texture slicing in a 1024x1024 viewport. The opaque surface in image (a) clearly shows block artefacts while these are difficult to perceive for the softer TF setting in image (c). Interblock interpolation (b & d) removes these artefacts. The II examples show single/dual pass framerate.

that suggests downsampling by skipping every second sample, arguably a less suitable approach.

2.10.3 Interblock Interpolation Results

The quality of the interblock interpolation sampling is shown in figure 2.24. A slightly rotated gradient is used to evaluate the different edge weight functions and the bottom row shows the pixel-wise error in CIE $L^*u^*v^*$ color space.

A final comparison between interblock interpolation and nearest block sampling is presented in figure 2.25. The top row, with a TF defining an opaque iso-surface, clearly shows block artefacts without II sampling. The softer TF setting for the bottom row shows less perceivable artefacts.

The multiresolution interblock interpolation scheme was initially deployed in a renderer based on texture slicing. The increased compu-

tational requirements for II sampling causes a significant performance reduction, about a factor of 8–10. It is, however, possible to render each slice in two passes. The first pass samples the interior domain and discards the fragment for samples outside the sample bounding box. The second pass then fills in the samples outside sample boundaries by exploiting early Z-termination⁴. The cost of the more expensive II sampling shader is thus significantly reduced and the performance hit is lowered to a factor of only 3–4, compared to NB sampling only. In this case it is important that the blocks at the lowest resolution are sampled entirely with NB sampling since the blocks with a single average value constitute a degenerate case with no interior intrablock sampling domain.

2.11 Raycasting on the GPU

The sampling techniques for flat multiresolution representations presented in previous sections are described using texture slicing techniques, but a more direct raycasting approach can be implemented on modern GPUs. The work presented there does not present increased performance for GPU-based raycasting over texture slicing techniques, but the rendering quality is improved and more complex rendering techniques are supported. In this section several techniques are described that significantly improve the rendering performance for GPU-based raycasting, as introduced in paper [63]. Exploiting the flexibility of programmable GPUs, several advanced classification techniques combining multiple TFs into single-pass raycasting, are briefly described. Full details are to be found in [67].

2.11.1 Adaptive Object-Space Sampling

As can be seen in images of volumetric data, there are often large areas of empty space and, for multiresolution volumes, blocks at less than full resolution. Obviously rendering performance could be increased if these parts were sampled more sparsely or skipped entirely. This has been the goal of many research efforts and several schemes have been proposed involving frame-to-frame and spatial coherence as well as using acceleration data structures such as octrees. These approaches have, in general, been mostly binary decisions, either a region is skipped or it is sampled at full density.

⁴Early Z-termination prevents the execution of a fragment program if the depth test fails.

Table 2.2: Rendering performance of the adaptive object-space sampling in a 1024×1024 viewport. Data reductions are given for each data set. TS refers to texture slicing and is comparable with full single-pass raycasting. Numbers specify FPS and (speed-up vs. full sampling). The last column indicates the expected theoretical speed-up.

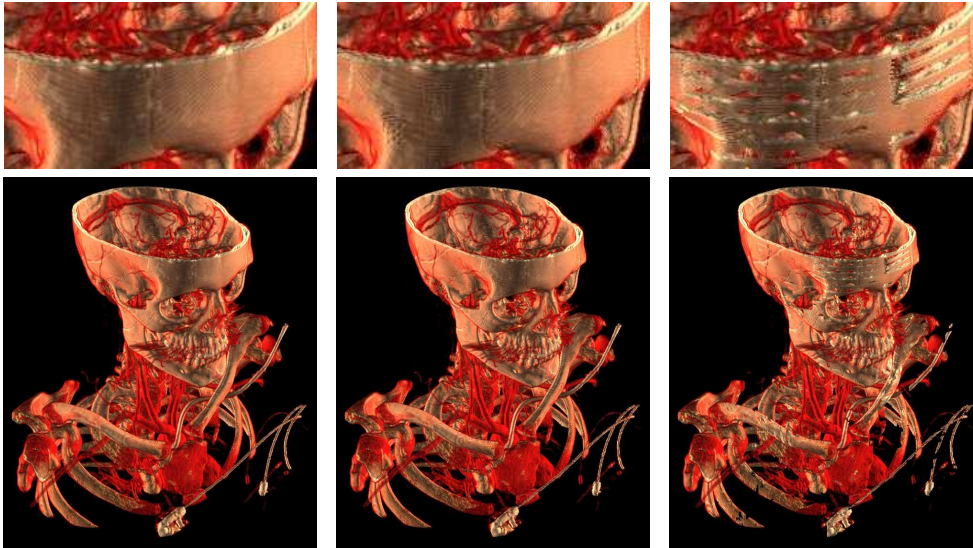
Dataset	GPU	TS	Full	Limited	Naive	Theoretical
Female 11:1	ATI	5.3	5.1	14.1 (2.8)	19.8 (3.9)	2.2
Female 11:1	NV	3.6	2.3	6.0 (2.6)	9.7 (4.2)	2.2
Female 30:1	ATI	5.5	5.2	17.4 (3.3)	27.5 (5.3)	3.1
Female 30:1	NV	3.6	2.3	6.5 (2.8)	11.1 (4.8)	3.1
Crab 9:1	ATI	3.4	2.9	6.2 (2.1)	8.7 (3.0)	2.1

Adaptive stepping along the ray was introduced in paper [63] where the block meta-data, the resolution level for each block, is used to adjust the density of samples. The LOD selection is, in fact, a fine-grained acceleration data structure with a more continuous adaptation directive, specifically when not rendering thin iso-surfaces only. The theoretical improvement through this scheme is a performance gain in proportion to the cube-root of the data reduction factor since the sampling is only reduced in one dimension, along the rays.

Special care has to be taken when stepping across block boundaries. In a naive approach, stepping is dictated by the resolution of the current block being sampled. A sample close to the block boundary could then potentially advance the ray deep into a neighboring block having a significantly higher resolution, resulting in an undersampled region. The scheme is therefore modified to limit the stepping at block boundaries. In paper [67] the raycaster was further developed to support interblock interpolation sampling, in this case the stepping is limited according to the minimum number of steps among all eight neighboring blocks.

Adaptive object-space sampling significantly improves the performance of GPU-based raycasting while maintaining a high visual quality. From table 2.2 the speed-up is slightly surpassing the expected theoretical speed-up (last column). The naive stepping approach shows even higher gains, clearly indicating that undesired undersampling occurs. The images in figure 2.26 show the result of full, limited and naive sampling density. The naive stepping shows obvious undersampling artefacts (fig. 2.26c).

Applying II sampling reduces the performance and attempts have



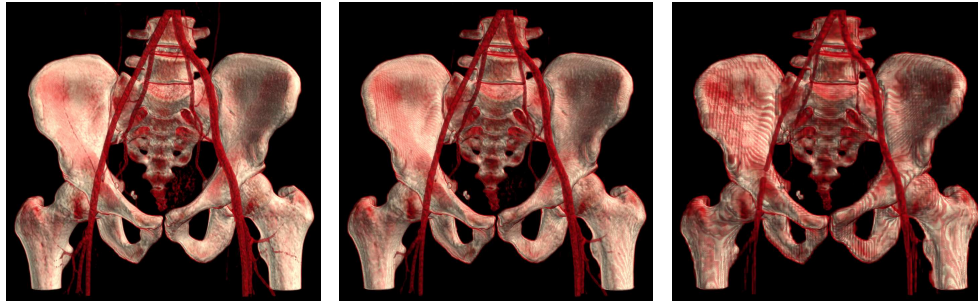
a) Full sampling (7.6 fps) b) Limited sampling (19.2 fps) c) Naive sampling (29.8 fps)

Figure 2.26: Adaptive sampling of multiresolution volumes. Single-pass, GPU-based raycasting enables efficient adaptation of sampling density along each ray individually. The multiresolution meta-data, the block scales, are used to define sampling density locally. Images are rendered in a 1024×1024 viewport.

been made to counter this effect by conditionally executing II sampling in the shader. Due to the SIMD nature of fragment processing units, processing many fragments in parallel, these efforts are often counter-effective in that fragments frequently do not behave uniformly over sufficiently large tiles. While this is an issue in the current GPU generation, the execution granularity is gradually being refined.

2.11.2 Flat Blocking Summary

The images in figure 2.27 capture several significant contributions of the flat blocking scheme, including interblock interpolation and adaptive sampling. They show the graceful degradation of reducing the volume data size to meet a progressively decreased memory budget. Image 2.27c uses only 2.7 MB of a 864 MB data set. Image 2.27a renders all blocks which contribute to the image at full resolution, referred to as virtually lossless. The fine-grained blocking scheme efficiently reduces the data size to 144 MB, a ratio of 6:1, without any loss of image quality.



a) 6:1, Virtually loss-less b) 80:1, Medium reduction c) 315:1, Extreme reduction

Figure 2.27: Transfer function based data reduction. This data set is first used in paper [66] which introduced perceptual TF-based level-of-detail selection and flat multiresolution blocking. They are here re-rendered using the final version of my GPU-based raycasting renderer.

Data reductions are maintained in all stages of the rendering pipeline, including texture memory on the GPU. Together with out-of-core data management this pipeline supports the rendering of data sets substantially larger than available core memory. This data reduction is achieved without including compression techniques, that could reduce data size in the first stages of the pipeline. Maintaining access to full resolution data at original precision is essential in the medical domain and the presented pipeline supports the use of full resolution data, given that the needed memory resources are available.

Course Notes

Advanced Illumination Techniques for GPU Volume Raycasting

Light Interaction

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Patric Ljung

Siemens Corporate Research, Princeton, USA

Christof Rezk Salama

University of Siegen, Germany

Timo Ropinski

University of Münster, Germany



SIGGRAPH2009

Light Transport and Illumination Models

In this chapter we discuss how light interactions can be computed for each voxel. Besides a brief overview of the Phong illumination with a focus on volume rendering, we will explain how to extend a raycaster to support shadowing as well as ray-traced reflections. The way light qualities change, when traveling through participating media is explained in the last chapter of this course.

3.1 Phong Illumination

The purpose of an illumination model, in both polygonal and volume rendering, is the simulation of real-world light interactions. During this simulation real-world phenomena as reflection and shadowing have to be incorporated. To improve the performance, often the simplified Phong illumination model is used, to reduce the $O(n^2)$ complexity, inherent to computing global illumination phenomena. By using this illumination model, only direct illumination is computed, i. e., illumination not influenced by other parts of the scene. Thus not every other part of the scene has to be considered when computing the illumination for the current object and thus the complexity is reduced to $O(n)$.

Also within volume rendering the Phong illumination model is often used to simulate light reflections. In the following we briefly describe its usage. We are not giving an entire introduction to the Phong model, and refer to [26] for further reading. For simplicity we assume that a volumetric data set is represented by a scalar intensity function which assigns to each point x a scalar value $f(x)$. In this context, the Phong computation is dependent of the following entities:

- Position of the current voxel x ,
- Gradient $\nabla\tau(f(x))$ at the current voxel position,


```

/**
 * Returns the diffuse term, considering the
 * currently set OpenGL lighting parameters.
 *
 * @param kd The diffuse color to be used.
 * Usually this is fetched from the transfer
 * function.
 * @param G The computed gradient.
 * @param L The normalized light vector.
 */
vec3 getDiffuseColor(in vec3 kd, in vec3 G, in vec3 L) {
    float GdotL = max(dot(G, L), 0.0);
    return kd * lightParams.diffuse.rgb * GdotL;
}

```

Listing 3.1: A simple GLSL shader, which computes the contribution of diffuse lighting.

- Voxel color as assigned through the transfer function, and
- Position of the light source.

To further simplify things, we assume for the following, that we have only *one* idealized point light source. We can compute the reflection occurring at the current voxel as a sum of three different illumination types: diffuse reflection, specular reflection and ambient lighting.

Diffuse reflections can be simulated by incorporating the Lambertian law, which states that the reflected light intensity is relative to the angle of incidence. Thus we can compute the diffuse reflection I_d for the current voxel by considering its normalized gradient $|\nabla \tau(f(x))|$ and the normalized light vector L as follows:

$$I_d(x) = L_{d,in} \cdot k_d \cdot \max(|\nabla \tau(f(x))| \cdot L, 0). \quad (3.1)$$

Thus, we can modulate the incoming diffuse lighting $L_{d,in}$ based on its incident angle and the current voxel color k_d . To achieve this effect within a volume raycaster the GLSL fragment shader shown in Listing 3.1 can be used.

In order to incorporate specular reflections, e.g., when dealing with specular materials as for instance metal acquired with a micro CT scanner, also the specular color for a voxel has to be computed. In contrast to diffuse reflections, specular reflections also depend on the viewing angle. Therefore the normalized view vector V is implicitly used to modulate the incoming specular lighting $L_{s,in}$ and the voxel's specular color k_s :

$$I_s(x) = L_{s,in} \cdot k_s \cdot \max(|\nabla \tau(f(x))| \cdot H, 0)^\alpha. \quad (3.2)$$

```

/**
 * Returns the specular term, considering the
 * currently set OpenGL lighting parameters.
 *
 * @param ks The specular color to be used.
 * @param G The computed gradient.
 * @param L The normalized light vector.
 * @param V The normalized view vector.
 */
vec3 getSpecularColor(in vec3 ks, in vec3 N, in vec3 L, in vec3 V) {
    vec3 H = normalize(V + L);
    float GdotH = pow(max(dot(G, H), 0.0), matParams.shininess);
    return ks * lightParams.specular.rgb * GdotH;
}

```

Listing 3.2: A simple GLSL shader, which computes the contribution of specular lighting.

α is used to influence the shape of the highlight seen on surface-like structures. A rather large α results in a small sharp highlight, while a smaller α results in a bigger smoother highlight. As it can be seen, V is not used directly, but the degree of reflection depends on the normalized half-way vector H , which can be computed as follows:

$$H = \frac{V + L}{2}. \quad (3.3)$$

For a detailed derivation of this equation, please also refer to [26]. To integrate specular reflections into a volume renderer, the GLSL shader shown in Listing 3.2 can be used. Please note, that since the half-way vector H is normalized, we can neglect the division by 2.

As stated above the Phong illumination model is a local illumination model, which considers only direct illumination, i.e., illumination coming from the light source directly, without being influenced by other objects. However, to approximate also indirect illumination effects, usually an ambient term is used. This proceeding ensures, that voxels with gradients pointing away from the light source do not appear pitch black. Since this ambient term does not incorporate any spatial information, it is the easiest to compute:

$$I_a(x) = L_{a,in} \cdot k_a, \quad (3.4)$$

by only considering the ambient light $L_{a,in}$ and the voxels ambient color k_a . The GLSL shader shown in Listing 3.3 can be used to incorporate the effect within a GPU-based raycaster.

Now, we know how to compute the three contributing illumination

```
/**
 * Returns the ambient term, considering the
 * currently set OpenGL lighting parameters.
 *
 * @param ka The ambient color to be used.
 * Usually this is fetched from the transfer
 * function.
 */
vec3 getAmbientColor(in vec3 ka) {
    return ka * lightParams.ambient.rgb;
}
```

Listing 3.3: A simple GLSL shader, which computes the contribution of ambient lighting.

types: diffuse, specular and ambient. Listing 3.4 shows how to combine them and thus realize the Phong illumination model.

To also incorporate attenuation based on the distance d of the current voxel to the light source, the computed light intensity can be modulated before returning it as shown in Listing 3.5. It requires a function for computing the attenuation, which is shown in Listing 3.6.

However, this distance based weighting does not incorporate any voxels possibly occluding the way to the light source, which would result in shadowing. To solve this issue, shadowing techniques need to be incorporated as described in Section 4 of this chapter.

So far, we have only considered a single light source. To extend the Phong illumination model to multiple light sources we can simply compute the contribution of each light source and add them up. This summation becomes possible, due to the additive nature of light energy.

3.2 Gradient Computation

As we have seen the gradient $\nabla\tau(f(x))$ is important for both, diffuse and specular reflections.

The quality of the computed gradients is crucial for the visual quality of the rendering. Especially when dealing with a high degree of specularity, it is important that the gradients change smoothly along a surface. In Figure 3.1 three gradient calculation methods are shown side-by-side: forward differences, central differences and Sobel gradients [95]. As expected, among these Sobel gradients result in the smoothest surfaces. This becomes directly visible when using the gradients for specular reflections (see Figure 3.6). While the visual difference is quite big when rendering artificial data sets, it is not as prominent when visualizing

```

/**
 * Calculates Phong shading.
 *
 * @param G The gradient given in volume object space (does not need to be
 normalized).
 * @param vpos The voxel position given in volume texture space.
 * @param kd The diffuse material color to be used.
 * @param ks The specular material color to be used.
 * @param ka The ambient material color to be used.
 */
vec3 phongShading(in vec3 G, in vec3 vpos, in vec3 kd, in vec3 ks, in vec3 ka) {

    vec3 L = normalize(lightPosition - vpos);
    vec3 V = normalize(cameraPosition - vpos);

    vec3 shadedColor = vec3(0.0);
    shadedColor += getDiffuseColor(kd, normalize(G), L);
    shadedColor += getSpecularColor(ks, normalize(G), L, V);
    shadedColor += getAmbientColor(ka);

    return shadedColor;
}

```

Listing 3.4: This GLSL shader realizes Phong shading by exploiting the previously introduced shaders.

```
shadedColor *= getAttenuation(d);
```

Listing 3.5: After computing the attenuation factor, it can be used to modulate the shaded color.

```

/**
 * Returns attenuation based on the currently
 * set OpenGL values. Incorporates constant,
 * linear and quadratic attenuation.
 *
 * @param d Distance to the light source.
 */
float getAttenuation(in float d) {
    return 1.0 / (lightParams.constantAttenuation +
        lightParams.linearAttenuation * d +
        lightParams.quadraticAttenuation * d * d);
}

```

Listing 3.6: Distance-based attenuation can be computed with a constant, a linear or a quadratic attenuation factor.

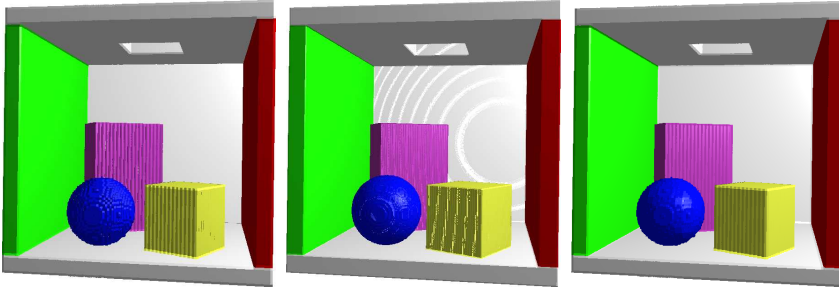


Figure 3.1: Visual comparison of different gradient computation methods when rendering a synthetic volume data set. From left to right: forward difference, central difference and Sobel gradients.

real-world data sets with smooth intensity changes. However, gradient filtering can be applied (see Figure 3.2) to further improve the rendering quality.

Due to its relatively high computing costs – 26 additional voxel lookups are needed – the Sobel operator is often applied in a preprocessing phase. This has the drawback, that the currently set transfer function does not have any effect on the computed gradients. Therefore, often forward and central differences are used, when rendering performance is important. As shown in Listing 3.7 computing forward difference gradients requires only three extra voxel lookups.

3.3 Specular Reflections through Ray-Tracing

Now, that we know how to compute the gradients, we can use them as normals in various ways to improve the illumination. For instance, the gradients can be used to realize a volumetric ray tracer. One approach for GPU-based raycasting considering only first order rays to be traversed has been proposed by Stegmaier et al. [97]. With their approach they are able to simulate mirror-like reflections and refraction. By casting a ray and computing its deflection, when a reflection or refraction occurs. To simulate the mirror-like reflection, they perform an environment texture lookup.

An alternative approach to trace rays of higher order, would be to use additional entry and exit points. This can be achieved by exploiting

```

/**
 * Calculate the gradient based on the A channel
 * using forward differences.
 */
vec3 calcGradient(sampler3D volume, vec3 voxPos, float t, vec3 dir) {
    vec3 gradient;

    float v = texture1D(transferFunc_, textureLookup3D(volume, volumeParameters,
voxPos).a);
    float v0 = texture1D(transferFunc_, textureLookup3D(volume, volumeParameters,
voxPos + vec3(offset.x, 0.0, 0.0)).a);
    float v1 = texture1D(transferFunc_, textureLookup3D(volume, volumeParameters,
voxPos + vec3(0, offset.y, 0)).a);
    float v2 = texture1D(transferFunc_, textureLookup3D(volume, volumeParameters,
voxPos + vec3(0, 0, offset.z)).a);

    gradient = vec3(v - v0, v - v1, v - v2);
    return gradient;
}

```

Listing 3.7: For computing forward difference gradients only 3 additional voxel lookups are required. By also fetching the transfer function, dynamic gradient changes are considered.

the workflow shown in Figure 3.3.

In the first stage the initial unmodified entry and exit points are computed as described in the first chapter of these course notes. By using these points the first order rays can be cast to generate one intermediate image as well as to construct a first hit point texture, i. e., the positions in volume space where a ray first encounters a visible medium (see Figure 3.4 (right)). While the intermediate image is cached in order to be able to blend it in the last stage, the first hit points are used as the entry points in the next recursion step.

To increase the foot print of the voxels encountered at the first hit point, the first hit point computation can be altered by sampling one step further into the volume. This ensures that the encountered medium is sufficiently penetrated and thus more clearly visible in the intermediate image. The exit points for the next recursion step can be computed in the second stage, as shown in Figure 3.3. Based on the entry position p and the direction d of a higher order ray r , the intersection between r and the bounding box of the volume can be computed. This is done by first performing an intersection point test for r and the six side planes of the bounding box. When knowing the normal N and the distance to the origin D of such a side plane, we can express each point x on the plane by the equation: $(N \cdot x) - D = 0$. By substituting the parameter form of the ray $x = p + t \cdot d$ into this equation, the following parameter value

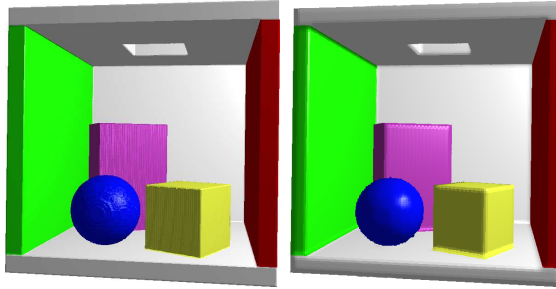


Figure 3.2: Gradient filtering improves the visual quality especially for synthetic data sets: central difference gradients (left), Sobel gradients (right).

for the intersection point can be obtained:

$$t' = \frac{D - (p \cdot N)}{d \cdot N}. \quad (3.5)$$

This parameter value needs to be computed for each of the six side planes, before choosing the smallest positive value t_{min} among these. After this computation, the entry and exit points can be forwarded to the subsequent raycaster. When all raycasters have finished rendering, the final image can be computed by blending all available intermediate images in stage 4 as shown in Figure 3.3.

As mentioned above, to compute the ray direction for rays reflected on a specular surface, the gradient $\nabla\tau(f(x))$ of this surface is considered. Similar to the computation of the specular reflection in the Phong illumination model as described in Section 3.1, the outgoing ray can be computed by considering the incoming ray and $\nabla\tau(f(x))$.

In cases where a refraction occurs, the incoming ray is bent at the surface based on Snell's law. Snell's law is used in physics to describe the behavior of a ray hitting an refractive interface, e. g., the surface of a glass-like object. It expresses the relationship between the angle of incidence ϕ and the angle of refraction θ and says that the ratio of the sines of ϕ and θ is a constant depending on the present media:

$$\frac{\sin\phi}{\sin\theta} = \frac{n_2}{n_1} \quad (3.6)$$

Thus, to compute the bending angle the refraction indices of the adjacent media, for which the refraction should be computed, has to be

Figure 3.3: Ray tracing can be achieved by using four subsequent stages. Initially the unmodified entry and exit points are computed for the first order rays (stage 1). Afterwards, for each recursive step, the exit points are calculated based on the first hit points of the previous step (stage 2). A raycaster computes intermediate images based on the generated entry and exit points (stage 3). Finally all intermediate images are blended into the final image (stage 4).

known. When assuming that these indices are n_1 for the medium which is left by the ray, and n_2 for the medium which is entered by the ray, we can compute the bending angle θ based on the incoming angle ϕ as follows:

$$\cos(\theta) = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \cdot (1 - (\cos(\phi))^2)}. \quad (3.7)$$

Thus, we can compute the direction of the leaving ray A_{vec} as follows:

$$A_{vec} = \left(\frac{n_1}{n_2}\right) E_{vec} + \left(\frac{n_1}{n_2} |\cos(\phi)| - \cos(\theta)\right) |\nabla \tau(f(x))|, \quad (3.8)$$

where E_{vec} is a vector representing the incoming ray.

In general total reflection has to be considered in cases the incoming ray hits the object under a very flat angle, i. e., no refraction but specular reflection occurs. The critical angle for which total reflection occurs can be computed as follows:

$$\phi_{crit} = \sin^{-1}\left(\frac{n_2}{n_1}\right). \quad (3.9)$$

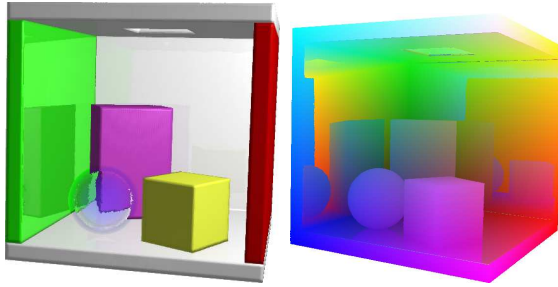


Figure 3.4: A synthetic Cornell box data set rendered with refraction and specular reflection (left) and its first hit points (right).

Thus, when the incident angle ϕ exceeds ϕ_{crit} , a specular reflection ray has to be computed instead of a refractive one.

The result of a ray tracer implemented using the concepts described above is shown in Figure 3.4 (left). As it can be seen in Figure 3.5 some scenes require only a rather low recursion depth to render them more realistically.

Again, by using different gradient calculation methods, different image qualities can be achieved as shown in Figure 3.6.

During the blending of the intermediate images in stage 4, a mapping function can be used, which expresses the specularity of certain intensity ranges, and thus influence the contribution of higher order rays to a pixels color. Alternatively, by using an automatic approach, the specularity can be set proportional to the length of $\nabla\tau(f(x))$. It should be mentioned that this is not an exact measure, since the gradient length determines only the intensity changes of neighboring voxels. However, when a higher intensity range occurs, one of the materials must have a higher intensity and may thus be assumed as being more specular.

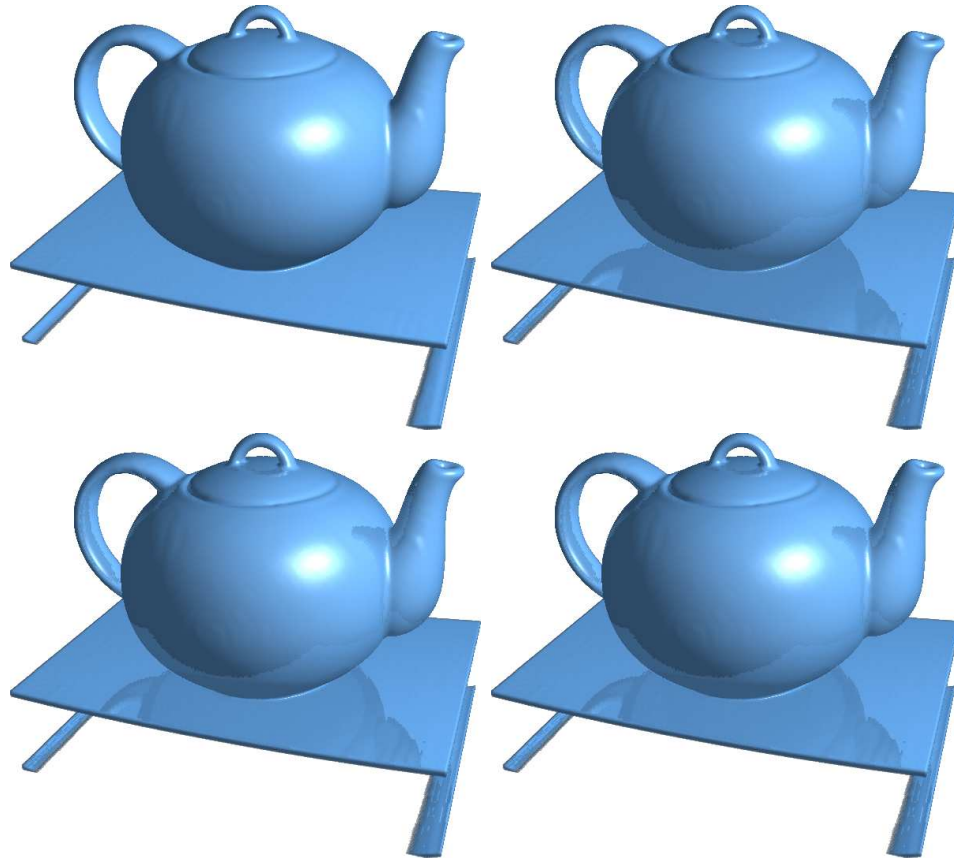


Figure 3.5: The teapot volume data set rendered with specular reflections. The images show renderings generated by traversing higher order rays: zero (top left) to up to three (bottom right).

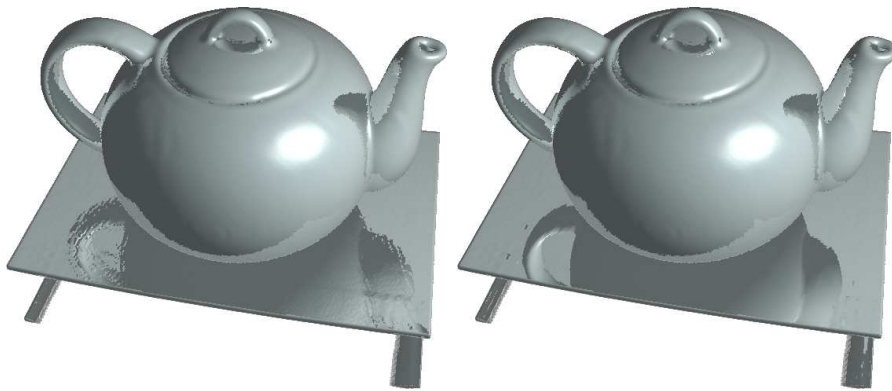


Figure 3.6: Specular reflections can be computed by incorporating forward difference gradients (left) and Sobel gradients (right).

Shadows

It has been shown that the used lighting model has a major impact on the spatial comprehension [58]. For instance, shadows serve as an important depth cue [89]. In this section we would like to discuss some techniques for integrating shadows into a GPU-based volume raycaster. While we will focus on volume rendering, it should also be mentioned that some researchers address the combination of polygonal and volumetric data when considering light interactions [113].

4.1 Soft vs. Hard Shadows

Various shadow algorithms have been developed in the field of computer graphics. Crow has proposed a shadow volume technique for generating shadows for scenes containing polygonal data [16]. To compute shadows of an object, its silhouette is extracted and extruded in direction of the light rays in order to generate shadow polygons which form the shadow volume of an object. During rendering each object is tested whether it lies inside or outside a shadow volume, and thus it can be determined whether the object is shadowed. Due to the polygonal nature of this algorithm, it is not suited for volume rendering. Another common approach for generating shadows when rendering polygonal data is shadow mapping which has been presented in 1978 by Williams [109]. Shadow mapping is an image-based approach which exploits an additional rendering pass in which the scene is rendered from the light source's point of view in order to determine the structures closest to the light source (see Figure 4.1). With this knowledge, a fragment-based shadow test can be introduced in the main rendering pass, i. e., each fragment is tested whether it is further away from the light source than the corresponding texel in the shadow map.

In contrast to the shadowing approaches for slice-based volume rendering [4, 52], only little has been done in order to integrate shadows into GPU-based raycasting. However, due to the similar ray paradigm, it should be noted that shadows have been integrated in volume ray-

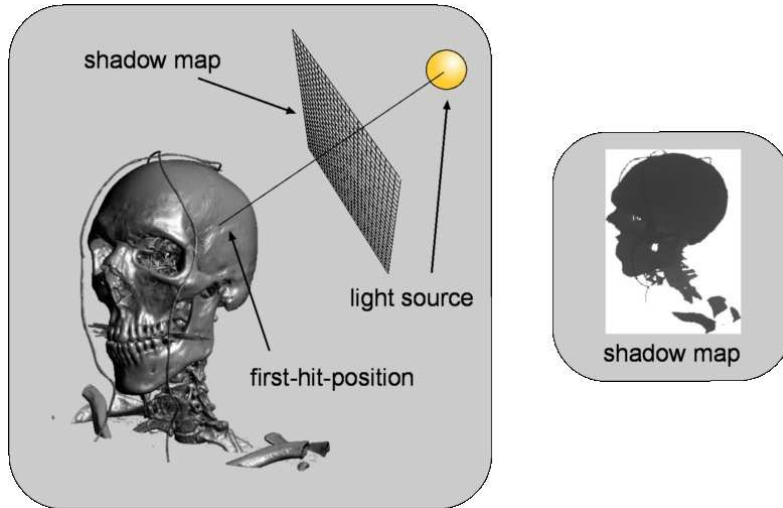


Figure 4.1: Shadow mapping exploits a single texture, which stores visibility information from the light source.

tracing systems [103, 102]. An overview of these techniques can be found in [72].

In analogy to the polygonal shadow mapping approach, for volume rendering also a depth map is needed in order to store light source visibility. Therefore first hit positions as seen from the light source can be computed and used to compute the light source distance (see Figure 4.2). When rendering polygonal models a depth value is properly defined. In contrast, volume rendering does not provide depth values in general. Therefore a shadow threshold value has to be used. With this threshold the intensity values representing opaque geometry can be changed, and with it the content of the generated shadow map. The example shadow map shown in Figure 4.2 has the same resolution as the viewport, i.e., 512×512 pixel, which allows to generate shadows without introducing aliasing artifacts (see Figure 4.7).

One benefit of shadow mapping is that soft shadows can be approximated by exploiting percentage closer filtering [82]. This is demonstrated in Figure 4.3, where the Visible Human torso data set is rendered with both, hard shadows (left) and fake soft shadows (right). Furthermore, when using shadow mapping the combination with polygonal models can be supported quite easily, since the geometry can also be represented within the shadow map.



Figure 4.2: The first hit points in color-coded volume coordinates as seen from the light source (left) and the resulting depth map (right). Both maps have a resolution of 512×512 texel and are generated during rendering the visible human head data set (see Figure 4.7).

4.2 Semi-Transparent Shadows with Deep Shadow Maps

While shadow mapping allows very efficient shadows on a fragment basis, it does not support semi-transparent occluders often occur in volume rendering. In order to address semi-transparent structures, opacity shadow maps serve as a stack of shadow maps, which store alpha values instead of depth values in each shadow map [50]. Another more compact representation for semi-transparent occluders are deep shadow maps [68]. The used data structure consists also of a stack of textures, but in contrast to the opacity shadow maps, an approximation to the shadow function is stored in these textures (see Figure 4.4). Thus it is possible to approximate shadows by using fewer hardware resources. Deep shadow mapping has first been applied to volume rendering by Hadwiger et al. [39].

While the original deep shadow map approach [68] stores the overall light intensity in each layer, in volume rendering it is advantageous to store the absorption given by the accumulated alpha value in analogy to the volume rendering integral. Thus, for each shadow ray, the alpha function is analyzed, i. e., the function describing the absorption, and approximate it by using linear functions.

To have a termination criterion for the fragment shader which performs the processing, the depth interval covered by each layer can be restricted. However, when it is determined that the currently analyzed

voxels cannot be approximated sufficiently by a linear function, smaller depth intervals are considered. Thus, the approximation works as follows. Initially, the first hit point for each shadow ray is computed, similar as done for the shadow mapping described above. Next, the distance to the light source of the first hit point and the alpha value for this position are stored within the first layer of the deep shadow map. Since we are at the first hit position, the alpha value usually equals zero when the shadow threshold is set accordingly. Starting from this first hit point, we traverse each shadow ray and check iteratively whether the samples encountered so far can be approximated by a linear function. When processing a sample, where this approximation would not be sufficient, the distance of the previous sample to the light source as well as the accumulated alpha value at the previous sample are stored in the next layer of the deep shadow map. This is repeated until all eight layers of the deep shadow map have been created.

To demonstrate the possibilities of deep shadow maps, we again use the simple volumetric Cornell box scene consisting of $128 \times 128 \times 128$ voxel (see Figure 4.5). The blue ball, which is set to be semi-transparent by modifying the transfer function, casts a *transparent* shadow on the back wall. The difference to the shadow of the opaque purple box can be noticed, especially when looking at the shadow borders. For generating the shown images, a deep shadow map consisting of eight layers

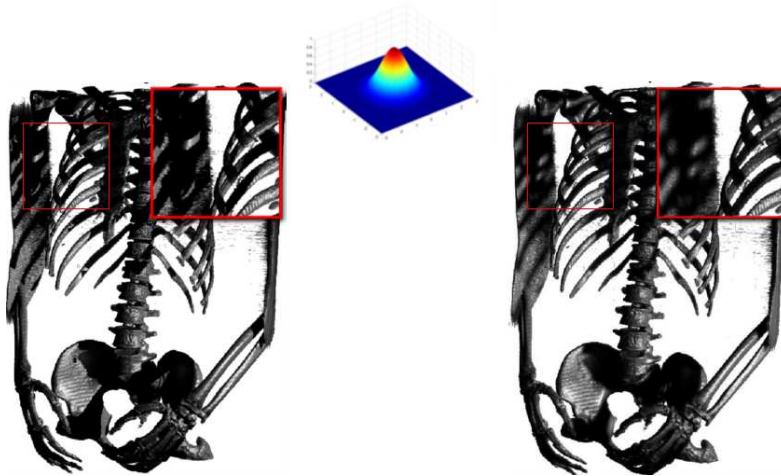


Figure 4.3: The Visible Human torso data set ($256 \times 256 \times 512$ voxel) rendered with hard shadows (left) and with fake soft shadows by using shadow mapping exploiting percentage closer filtering (right).

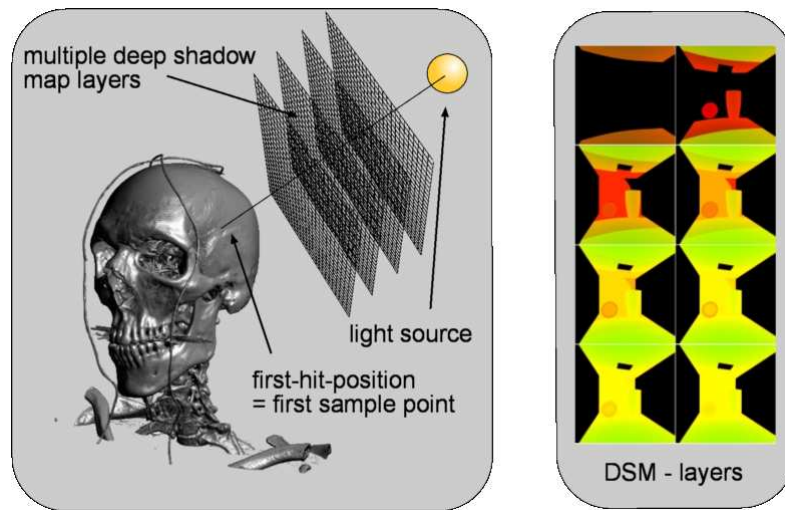


Figure 4.4: Deep shadow mapping exploits a stack of textures, which stores an approximation to the shadow function.

has been used. A color coded version of the layers is also shown in Figure 4.5 (bottom). To generate the layers rapidly, current graphics hardware can be exploited. By using multiple render targets eight layers could be computed in a single rendering pass. However, using only four rendering targets and creating eight deep shadow layers by performing simple channel splitting allows a more efficient storage. Therefore, the light source distance and the alpha value are stored in two successive channels, i.e., R and G as well as B and A . Thus, two shadow layers can be represented by using only one $RGBA$ texture. However, for illustration purposes, we wrote these values into the R and G channels when generating the pictures shown in Figure 4.5 (bottom).

When generating the deep shadow maps, an error value is introduced to determine whether the currently analyzed samples can be approximated by a linear function. In analogy to the original deep shadow mapping technique [68], this error value constrains the variance of the approximation. This can be done by adding (resp. subtracting) the error value at each sample's position. When the alpha function does not lie within the range given by the error value anymore, a new segment to be approximated by a linear function is started. The effect of choosing a too small error value is shown in Figure 4.6. As it can be seen, a too small error value results in a too close approximation, and the eight layers are not sufficient anymore to represent shadow rays having a higher depth

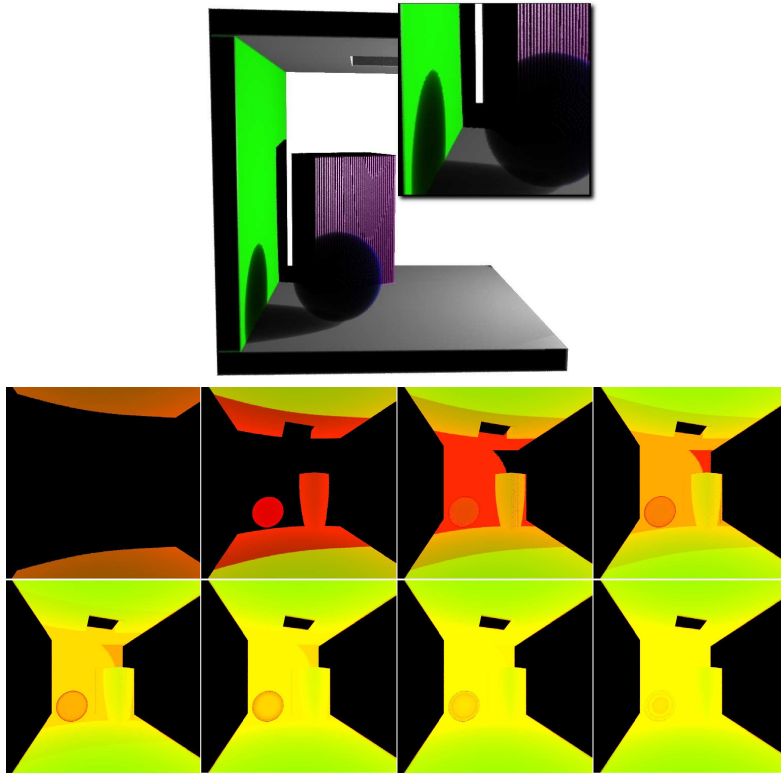


Figure 4.5: A synthetic scene ($128 \times 128 \times 128$ voxel) rendered using deep shadow mapping. The shadow of the semi-transparent blue ball is correctly captured. The images on the bottom show the successive layers of the deep shadow map.

complexity. Thus especially in regions, where the two occluders both intersect a shadow ray, shadow artifacts appear. Obviously this drawback can be avoided by introducing additional shadow layers. This would allow a more precise approximation of the shadow function, but would also result in decreased rendering performance since additional rendering passes are required.

A comparison of different shadow computation methods is shown in Figure 4.7 and Figure 4.8. As it can be seen deep shadow maps introduce artifacts when thin occluder structures are present. The shadows of these structures show transparency effects although the objects are opaque. This results from the fact that an approximation of the alpha function is exploited. Especially thin structures may diminish when approximating over too long depth intervals.

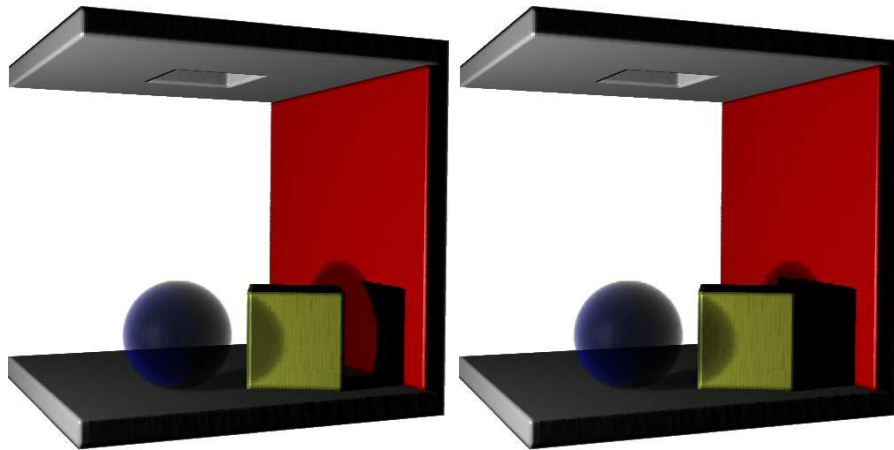


Figure 4.6: Different error values for deep shadow mapping: 0.00005 (left), 0.01 (right). Artifacts appear when using too small error values.

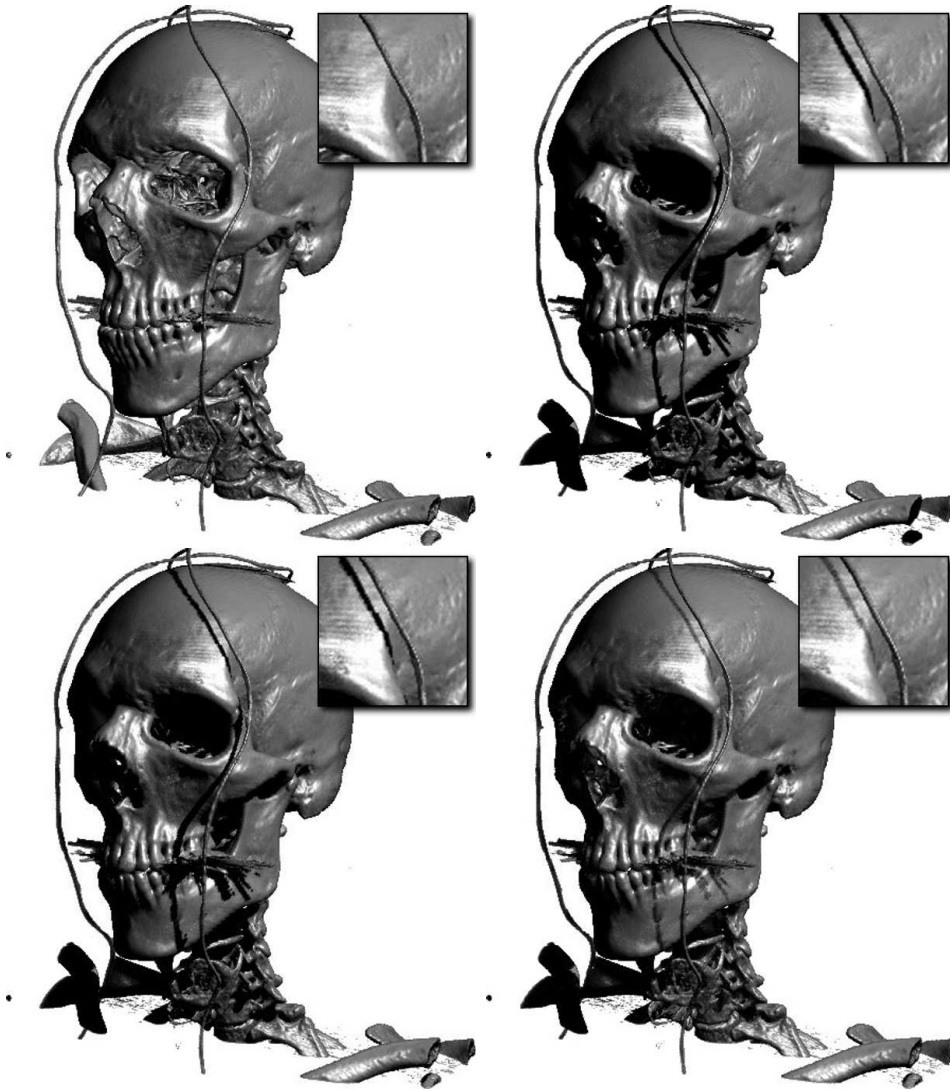


Figure 4.7: The Visible Human head data set ($512 \times 512 \times 294$ voxel) rendered without shadows, with shadow rays computed by a ray tracer, with shadow mapping and with deep shadow maps (from top left to bottom right). The generated shadow map is shown in Figure 4.2.

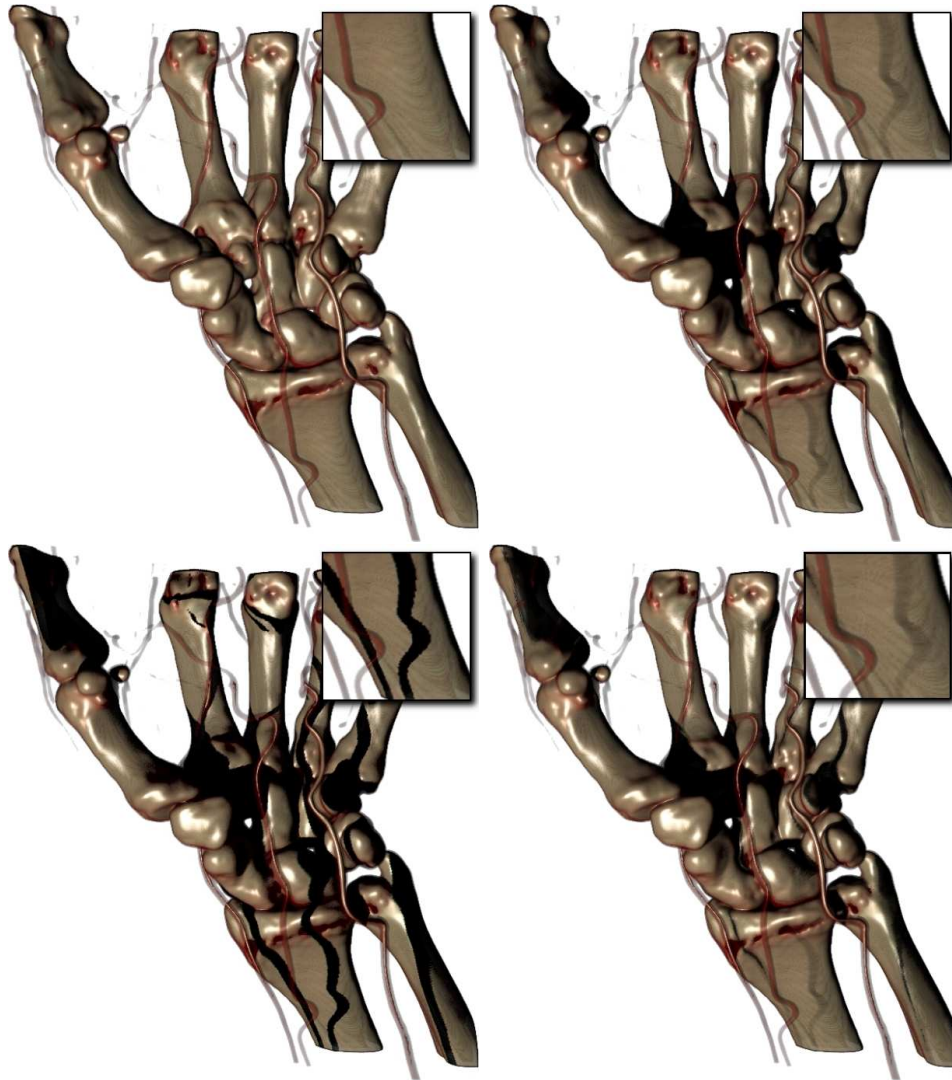


Figure 4.8: The hand data set ($244 \times 124 \times 257$ voxel) rendered without shadows, with shadow rays, with shadow mapping and with deep shadow maps (from top left to bottom right). Semi-transparent shadows become visible when using shadow rays or deep shadow maps.

Course Notes

Advanced Illumination Techniques for GPU Volume Raycasting

Ambient Occlusion

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Patric Ljung

Siemens Corporate Research, Princeton, USA

Christof Rezk Salama

University of Siegen, Germany

Timo Ropinski

University of Münster, Germany



SIGGRAPH2009

Ambient Occlusion for Isosurfaces

Ambient Occlusion refers to techniques to simulate global lighting by estimating the visibility of a global ambient omnidirectional illuminant from a primitive, be it a polygon vertex, surface location or voxel in a volume. In this section we will outline various methods based on volumetric models or data sets. We have divided the algorithms into two classes; isosurface based volume rendering and full volume rendering, incorporating semi-transparent samples and translucency. In both cases direct volume rendering is considered, that is, no intermediate geometry is created for isosurfaces. The remainder of this chapter describes techniques for direct isosurface volume rendering.

In the past many methods based on the theory of light transfer have been developed to enable interactive diffuse interreflections for polygonal models [93, 11]. Most of these physically motivated approaches are based on pre-computing illumination for all vertices and storing it in an appropriate data structure which is accessed during rendering. Thus these algorithms support interactive modification of light and camera parameters as well as some material parameters. However, the application to deformable geometry is constrained and requires a new pre-computation in most cases [94]. To address these limitations, approximations have been proposed, which are not physically motivated, but lead to visually convincing results. In contrast to polygonal models the structure represented by a volumetric data set depends on the rendering parameters, which include the transfer function as well as thresholding parameters, since these rendering parameters can be used to omit certain voxels from being displayed. For instance, by only changing the transfer function or the thresholding, a medical volume data set can be visually mapped to many distinct structures, e. g., the skeleton only, the muscles only or the skeleton with surrounding tissue. Obviously this structural variance has a strong impact on the light interaction between the structures of such a data set, rendering the currently known surface-based illumination techniques insufficient for interactive volume illumination. Since trans-

fer function as well as thresholding are altered frequently it should be possible to perform these changes interactively [51]. Another challenge is the fact that volume rendering requires to compute light interactions for several samples along a viewing ray and thus introduces a higher level of complexity compared to computing light interactions only once for each fragment as is necessary when rendering opaque polygonal data.

Vicinity Shading [98] simulates illumination of isosurfaces by taking into account neighboring voxels. In a pre-computation the vicinity of each voxel is analyzed and the resulting value, which represents the occlusion of the voxel, is stored in a shading texture which can be accessed during rendering. Vicinity Shading requires a new preprocessing when changing the rendering parameters, and it does not support color bleeding.

Desgranges and Engel describe a less expensive approximation of ambient light than Vicinity Shading [20]. They combine ambient occlusion volumes from different filtered volumes into a composite occlusion volume. While pre-processing time is greatly reduced the ambient occlusion volume still must be recomputed whenever the transfer function is changed.

Wyman et al. have presented a technique to pre-compute or lazily compute global illumination for interactive rendering of isosurfaces extracted from volumetric data sets [111]. They support the simulation of direct lighting, shadows and interreflections by storing pre-computed global illumination in an additional volume to allow viewpoint, lighting and isovalue changes. Beason et al. present a method which additionally can represent translucency and caustics but supports static lighting only [3]. For that purpose they extract different isosurfaces from a volumetric data set, illuminate them with a path tracer and store the results in a new volume data set. During rendering they can interactively change the isovalue and access the pre-computed illumination. Penner and Mitchell recently proposed a method based on histograms to classify the visibility around a voxel [80]. It bears some similarities with the technique by Ropinski et al. [85], described later.

All these surface illumination models are only applicable to isosurfaces representing a single intensity within the data set, but do not allow to consider multiple surfaces corresponding to different intensities. Hence it is not possible to represent the entire volume data set, whereas varying intensities are one of the major advantages over polygonal models, e. g., when representing different types of tissue.

Ambient Occlusion for Direct Volume Rendering

Desgranges and Engel describe a less expensive approximation of ambient light than Vicinity Shading [20]. They combine ambient occlusion volumes from different filtered volumes into a composite occlusion volume. While pre-processing time is greatly reduced the ambient occlusion volume still must be recomputed whenever the transfer function is changed.

6.1 Local Ambient Occlusion

Recently Hernell et al. have proposed a method for computing ambient and emissive tissue illumination efficiently [42]. This technique is based on shooting rays in several directions from each non-transparent voxel in the data set. The rays are terminated at predefined radial boundary making this technique a local approach. Since these rays are generated for each non-transparent voxel in the data set it benefits from multiresolution level-of-detail data reduction. Empty space is ignored and low resolution regions require less processing. This technique has been implemented using the multiresolution techniques by Ljung et al. [63, 65, 67]

In many applications of volume rendering it is not desired to create a realistic global illumination where objects can be fully shadowed. A local model is therefore proposed, where a limited spherical neighborhood, Ω , around each processed voxel is evaluated. To consider semi-transparent materials the local ambient occlusion model needs to go beyond the all-or-nothing approach used in previous work. The proposed technique includes semi-transparent objects and supports a smooth fading of shadows as occluding objects occupy more or less of the local neighborhood. The scheme is based on sampling the locally occluding material along multiple rays originating from the voxel at the center of Ω .

The incident light intensity, $I_k(x)$, arriving at a voxel location, x ,

from one ray direction, k , is given by the equation:

$$I_k(x) = \int_a^{R_\Omega} \frac{w_k}{R_\Omega - a} \exp\left(-\int_a^s \tau(u) du\right) ds \quad (6.1)$$

where a is an initial offset from the voxel along the ray and R_Ω is the radius of the spherical support. A ray is also associated with a weight, w_k , and thus enables directional weighting of the ambient light. The attenuation of light contribution along the ray is estimated using the optical depth, the integral of transfer function densities, $\tau(s)$. Numerical evaluation of the integral in equation 6.1 is obtained as:

$$I_k(x) = \sum_{m=0}^M \frac{w_k}{M} \prod_{i=0}^{m-1} (1 - \alpha_i) \quad (6.2)$$

in a front-to-back compositing scheme, where M is the number of samples along the ray and α_i is the sample's opacity according to the current transfer function.

Light is contributed at each sample point, normalized to sum to one. This ensures that shadows from an occluding object increase smoothly as a larger fraction of the ray penetrates the occluding object. This effect can be seen in figure 6.3 (top-right). The local ambient occlusion for a voxel, $I(x)$, is then given by the sum of all incident light rays I_k

$$I(x) = \frac{1}{K} \sum_k^K I_k(x). \quad (6.3)$$

Figure 6.3 (bottom-right image), shows 64 rays. The directions for the rays are created by subdividing a tetrahedron, icosahedron or octahedron to different levels, depending on how many rays are desired.

Estimation of the volumetric Local Ambient Occlusion (LAO) is performed as an initial step of the rendering pipeline. The resulting light intensity of each voxel is stored in an intensity map which is used in the final volume rendering to illuminate each sample.

To further control the effect of the ambient occlusion the final value can be adjusted according to a gamma function. In addition, an ambient bias can be specified to adjust the amount of minimum lighting. The final luminance, $I'(x)$, is computed by the following equation.

$$I'(x) = (I_{\text{bias}} + I(x))^\gamma \quad (6.4)$$

6.1.1 Emissive Tissues and Local Ambient Occlusion

Having established a framework for volumetric local ambient occlusion we can proceed to incorporate emissive components from the transfer function. This simulates single light scattering of luminous objects within the volume. Equation 6.1 is simply modified to also incorporate colored light emission, c_E .

$$I_k(x) = \int_a^{R_\Omega} w_k \frac{1 + c_E(s)}{R_\Omega - a} \exp\left(-\int_a^s \tau(u) du\right) ds \quad (6.5)$$

The emittance of a sample point along a ray within Ω can therefore affect the intensity and color of point x . The emissive parameter is included during the final ray-casting as well, further enhancing the effect of luminosity.

6.1.2 Integrating Multiresolution Volumes

Mapping from volume coordinates to packed coordinates is straightforward using the forward index texture mentioned above, described in more detail in [65]. The reverse mapping presents a minor challenge, since the location of a voxel in the volume must be determined. Potentially a block could be represented by a single voxel in the packed texture which would require a reverse index map of the same size as the packed texture itself. This is avoided by ignoring reverse mapping of blocks below a certain minimum resolution level.

By ignoring blocks with the two lowest resolution levels, 1^3 and 2^3 , the size of the reverse index map can be reduced by a factor of 64. All empty blocks are assigned the lowest resolution or ignored entirely and the remaining blocks are assigned higher resolutions, with a minimum of 4^3 voxels. The reverse map can then hold a block's location in volume coordinates (V) for all non-empty blocks. Figure 6.1 illustrates both the forward and reverse mapping schemes.

Neighboring blocks in the packed texture are not always located closely in the original volume. Since linear interpolation is used between texels this implies that interpolation also occurs between block boundaries which then leads to artifacts. A distance δ is therefore used to clamp the coordinates so that interpolation only occurs within each block [65]. In the local occlusion calculation interpolation of samples between blocks is not considered.

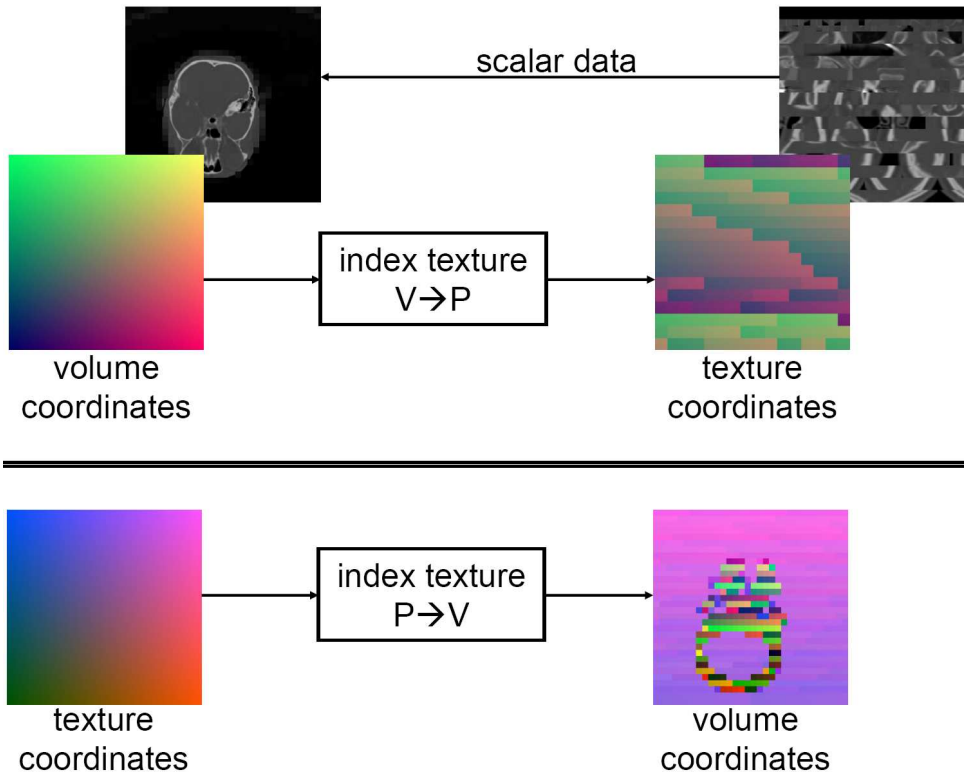


Figure 6.1: Coordinate mapping between the volume and packed texture. Volume coordinates are mapped via the index texture to packed coordinates. Packed texture coordinates are similarly transformed, but via the reverse index texture.

6.1.2.1 Multiresolution Ambient Occlusion Pipeline

The LAO computation is driven by the processing of each voxel in the volume. Since a multiresolution volume management is applied the gained data reduction from the LOD selection also implies a speed-up of the LAO processing. With increased data reduction the LAO processing is also reduced, the LAO illumination is computed with the same data reduction as the LOD selection of the volume. The processing of each voxel can, in turn, be further accelerated by adapting the sampling rate depending on the voxel's block resolution level. This per-fragment processing pipeline is presented next.

The pipeline is executed for all voxels in a slice of the packed volume texture in parallel. A 3D texture is created to hold both the scalar values of the volume and the colored illumination of each voxel. The entire volume is then processed by mapping all the slices, one-by-one,

as a rendering target of a framebuffer object. The fragment pipeline is initiated by rendering a large quad over the entire framebuffer, where each pixel maps to one voxel in the mapped 3D texture. This approach is efficient but requires that the hardware support the attaching of a slice of a 3D texture to a framebuffer, as available in the latest GPU hardware generation, e.g. NVIDIA GF8800.

For each voxel a fragment program is initiated. An outline of the per-fragment processing is shown in 6.2. The fragment location is used to lookup the corresponding voxel location in the volume through the reverse index texture, $P \rightarrow V$. In addition a per-slice constant z-position is provided to find the correct 3D location. The pipeline then iterates over all the incident light directions in Ω . The number of rays, K , to sample can be dynamically configured and the ray directions are stored in a texture together with the directional weight, w_k .

The sampling along a ray starts at a user-defined offset, a . In addition the sampling density, d , and radius, R_Ω , of Ω can be specified. The initial offset avoids unnecessary self-occlusion for voxels on the boundary of highly opaque regions. The sampling density is adjusted for voxels in lower resolution blocks. A voxel belonging to a block of resolution level, λ , relative to the highest resolution level, increases the sample distance, the step length, by a factor of 2^λ . Thus, the sampling rate stays fixed relative to the sampling density of the underlying volume data. The immediate lower resolution will have twice the sampling distance. When the sampling rate changes the opacities and colors have to be corrected as well, this is done on the fly using the opacity correction formula:

$$\alpha_{\text{mod}} = 1.0 - (1.0 - \alpha_{\text{orig}})^{2^{\lambda d}}. \quad (6.6)$$

6.1.3 Adding Global Light Propagation

Hernell et al. [43] recently extended the Local Ambient Occlusion approach described above to include a Global Light propagation and first order scattering effects. Below we describe this approach. Figure 6.7 shows a schematic overview of the technique.

The method progresses in several steps to simulate the light transport in the volume and, at each step, captures the main physical contributions. In the first step opacities are composited into a global shadow volume for a given light source and transfer function (TF) settings. The calculation is based on piecewise integration techniques on the GPU and sparse representations of intermediate results. The method then proceeds to include first order scattering of the global light arriving at each voxel by in-

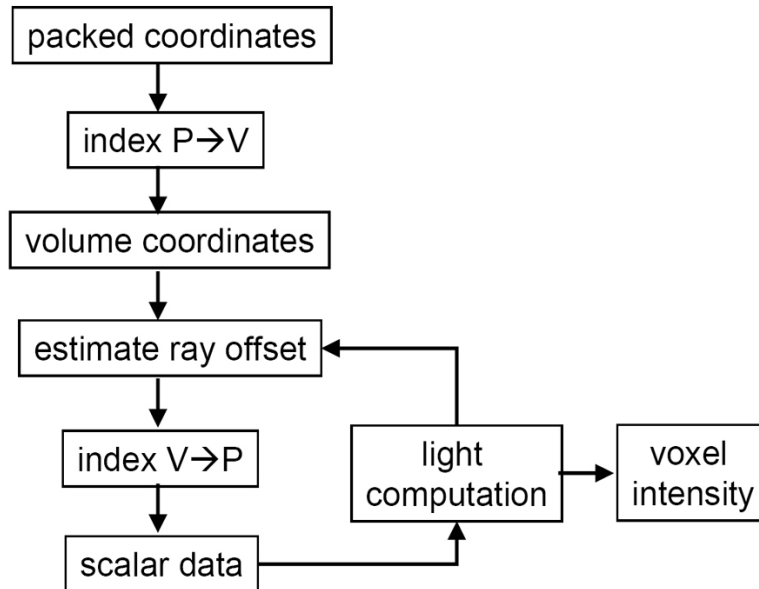


Figure 6.2: Per-fragment computations to evaluate LAO for a voxel. The packed coordinates (P) are transformed to volume coordinates (V). Rays are generated to sample the volume around the voxel. For each sample the reverse index texture gives locations in the packed volume. Final fragment intensity is the average of the ray intensities.

tegration of scattered light in a local spherical neighborhood around each voxel. This step is similar to the local ambient occlusion (LAO) method described in [42], which enhances perception of local shapes and tissue properties. In the last step a single pass ray-casting approach is used to render the final image. The resulting application enables updates of light position and transfer functions while maintaining interactive frame rates yet simulating a realistic light model.

6.2 Dynamic Ambient Occlusion

In this section we cover recent techniques based on Ropinski et al. [85] which realizes dynamic ambient occlusion as well as an approximation to color bleeding when rendering volumetric data sets. The method is independent of the currently applied transfer function as well as the thresholding and therefore represents ambient occlusion as well as color bleeding like effects for all combinations of structures contained in a



Figure 6.3: The image in the top left shows diffuse illumination while the other three show local ambient occlusion using a single ray directed at the single light source (top right), 8 rays (bottom left) and 64 rays (bottom right). The LAO approach generates convincing shadowing effects giving a superior 3D appearance



Figure 6.4: Emissive illumination effects. Left to right: Emissive light captured in the ray-tracing stage, emitted light illuminating the surrounding material, and both effects combined.

Data reduction	A		B	C	Piecwise segment length (voxels)	A			
	32 ³	256 ³				32 ³	64 ³	128 ³	256 ³
8.9:1	284	552	233	68	4	261	267	439	1428
14.8:1	178	515	145	68	8	284	297	373	552
22.1:1	121	403	96	48	16	331	339	380	641
35.2:1	81	365	62	46	32	436	439	463	862

Table 6.1: Performance, in milliseconds (ms), for different levels of illumination updates ((A), (B) and (C)) versus varying data reductions (left table). The piecewise segments are 8 voxels long. In the right table the performance is shown for different sizes of the SVR versus varying lengths of the local piecewise segments for light update (A). The data reduction is 8.9:1. Measurements in both tables are performed for a volume of 512^3 voxels, rendered in a 1024×1024 window. The same volume, TF and rendering settings are used in fig. 6.10, using a step length of 16 voxels.

volume data set, which can be extracted interactively by changing these rendering parameters. However, the presented rendering algorithm is not based on the physics of light propagation, but provides a visually convincing approximation. It can be applied to direct volume rendering (DVR) as well as isosurface shading techniques, and for the latter the isovalue can be changed interactively. Rendering time is kept low, since the proposed technique requires only little overhead compared to the solution of the standard volume rendering integral combined with the application of strictly local illumination. Besides the transfer function and the thresholding, lighting as well as the camera parameters can be changed interactively. Furthermore, in contrast to frequently used surface-based illumination models our technique does not necessarily require a gradient calculation and is therefore also applicable to homogeneous regions.

In the pre-computation we ensure that we analyze and store the environment of each voxel x in such a way that we are able to compute an environmental color E_{env} — approximating the influence of the voxels neighborhood — during rendering interactively. In order to support interactive modification of the transfer function and the thresholding, the computation is performed independently of these rendering parameters. The consecutive steps needed are shown in Figure 6.12 and are further explained in the following subsections.

6.2.1 Local Histogram Generation

To approximate the environmental color E_{env} for a given voxel x , we exploit its local histogram $LH(x)$. Local histograms have also been used in other areas of volume rendering [86, 71]. For our approach local his-

tograms are adequate, because indirect illumination can be calculated properly for a given point by considering close objects only [18]. All voxels \tilde{x} lying in a sphere $S_r(x)$ with radius r centered around x contribute to the local histogram $LH(x)$, weighted based on their distance to x (see step 1 in Figure 6.12). Thus assuming that $f(x) \in [0, 2^b]$, with $b \in \{8, 12, 16\}$ being the bit depth of the data set, LH assigns to each x an n -tuple, with $n = 2^b$:

$$LH(x) = (LH_0(x), \dots, LH_{n-1}(x)), \text{ with} \quad (6.7)$$

$$LH_k(x) = \sum_{\substack{\tilde{x} \in S_r(x) \\ \tilde{x} \neq x}} f_{dist} \left(\frac{|x - \tilde{x}|}{d_{min}} \right) \cdot g(f(\tilde{x}), k). \quad (6.8)$$

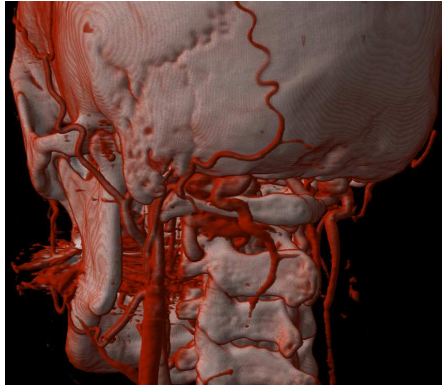
d_{min} denotes the minimal distance between any two different voxels in the data set. $f_{dist} = \frac{1}{d^2}$ is used to achieve a distance based weighting and takes into account that energy falls off as the inverse square of the distance. $g(i, k)$ is used to group the intensity values appropriately:

$$g(i, k) = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{otherwise.} \end{cases} \quad (6.9)$$

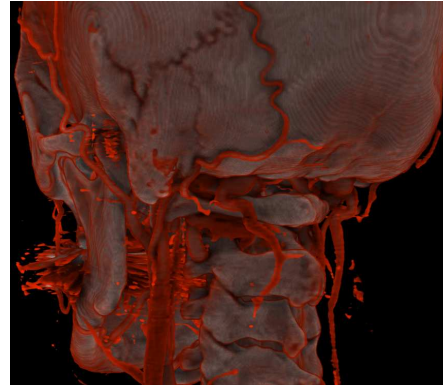
$LH_k(x)$ represents the influence voxels of intensity k in the neighborhood of x have on x . In contrast to other techniques we can not consider the attenuation of light which results from voxels lying between the current voxels and its neighbors, because we discard the spatial locations. Since we are only interested in the relative distribution of $LH(x)$, we normalize the values in each $LH(x)$ with respect to the number of voxels lying in the sphere $S_r(x)$ with radius r centered around x .

To capture the neighborhood of an object in scenes consisting of polygons, often ray casting is exploited which involves sampling that may influence the image quality. Since volume data sets are already a discretized representation, ray casting and thus possible sampling artifacts should be avoided. We use a simple method which captures the vicinity of voxel x by iterating over all voxels \tilde{x} in its neighborhood and adding their contribution to $LH(x)$ as defined by equation (6.8). By definition $LH(x)$ does not contain any spatial information except the distance based weighting f_{dist} inherently capturing the degree of influence of the voxels in the neighborhood. In order to capture also directional information, we subdivide $S_r(x)$ based on the gradient at x into two hemispheres. Instead of one local histogram for $S_r(x)$, we compute two local histograms, one for the forward facing hemispherical region $H_f(x)$ and one for the backward facing hemispherical region $H_b(x)$ (see Figure 6.12).

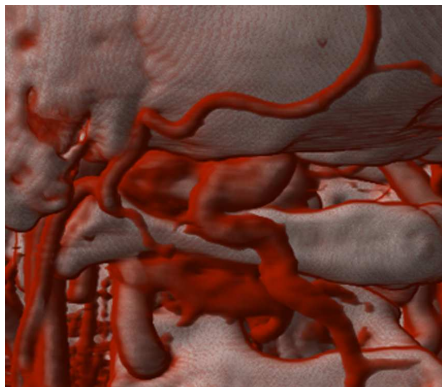
As already mentioned, instead of capturing the light interactions between all voxels of a data set, we consider only the vicinity defined by the radius r . Obviously r is data set dependent, but can generally be chosen rather small in comparison to the number of voxels n . This reduces the complexity from $O(n^2)$ operations to $O(r^3 \cdot n)$.



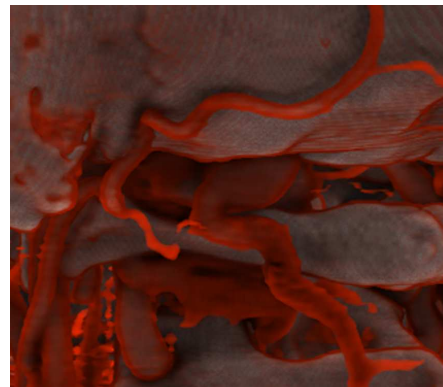
a) Diffuse Illumination



b) Local Ambient Occlusion, 32 rays



c) Close-up of vessels in (a)



d) Close-up of vessels in (b)

Figure 6.5: Example images showing the enhanced 3D structure made clear through the LAO method.



(a) Diffuse Illumination (b) LAO with illuminating material (c) LAO with emissive ray-tracing

Figure 6.6: Example images showing the enhanced information from the emissive materials. The bullet and fragments are clearly visible in the abdomen. The effect of the LAO in revealing the bone structure is also very clear.

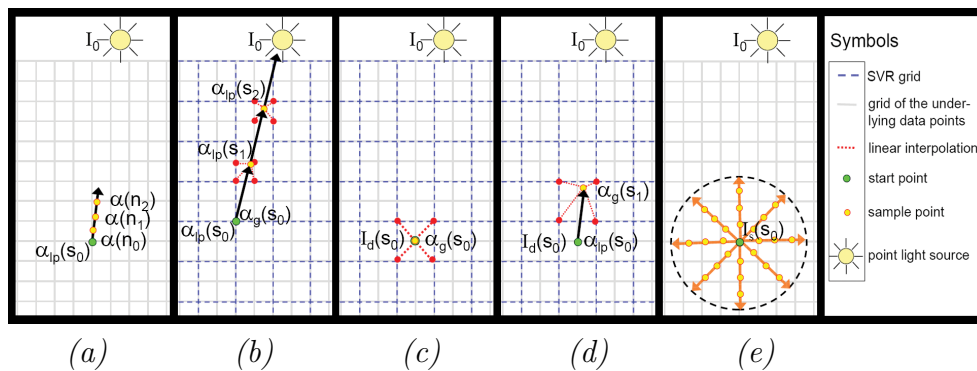


Figure 6.7: Opacity for a local segment is integrated in (a) and global opacity is integrated in (b). A simple method to compute I_d is to perform a direct interpolation from α_g , as in (c), however, a better approximation is obtained, in (d), by utilizing the piecewise integration as an initial step. The quality of nearby shadow contributions is thereby increased. In-scattering of the initial intensity approximation, I_s , is illustrated in (e).

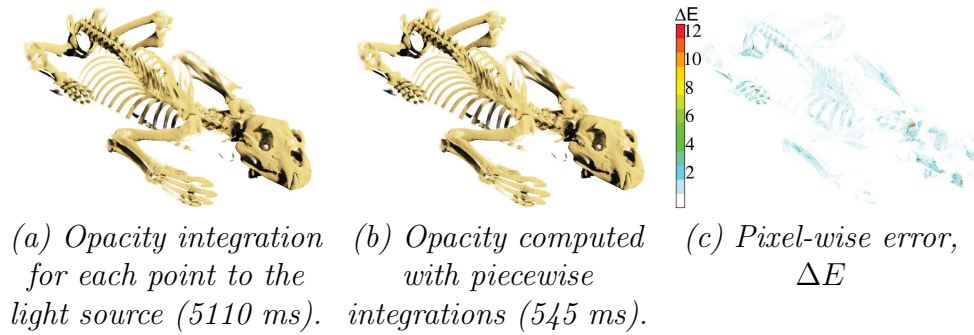


Figure 6.8: A comparison of illumination integrated for each point to the light source, in (a), with intensities computed with our approximation using piecewise integration, in (b). The volume and the SVR are 512^3 voxels, the view-port is 1024×1024 pixels and no in-scattering is considered. A color-coded error image that shows the pixel-wise error, ΔE , is provided in (c). The errors that appear are small ($\Delta E_{RMS} = 0.19$ and $\Delta E_6 = 6\%$), and mainly appear at sharp edges and thin structures.

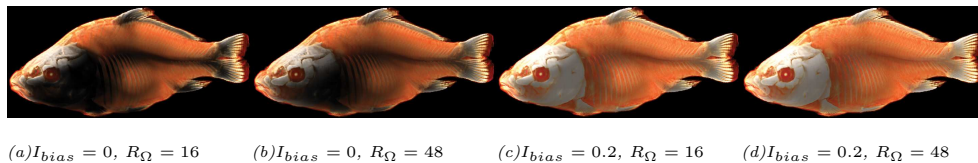


Figure 6.9: Light is integrated, for a CT scan of a carp, with varying settings for I_{bias} and R_{Ω} . The shadows becomes less distinct with a large radius (denoted in units of voxels) of the scattering sphere, Ω . I_{bias} adjusts the minimum intensity of each voxel.

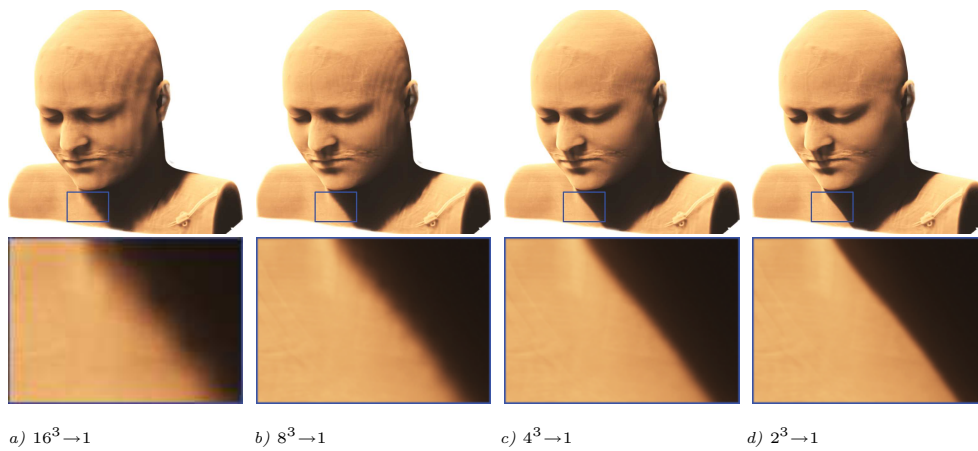


Figure 6.10: Block artifacts appear due to reduced size of the shadow volume representation. The original volume size is 512^3 with a data reduction of 8.9:1 and the SVR is computed for (a) 32^3 (b) 64^3 (c) 128^3 and (d) 256^3 voxels. Jagged edges appear in (a) and (b), as can be seen in the close ups. Also, band artifacts arise at the side of the head. Computation times for these images are as shown in table 6.1. The length used for the piecewise segments is 16 voxels.

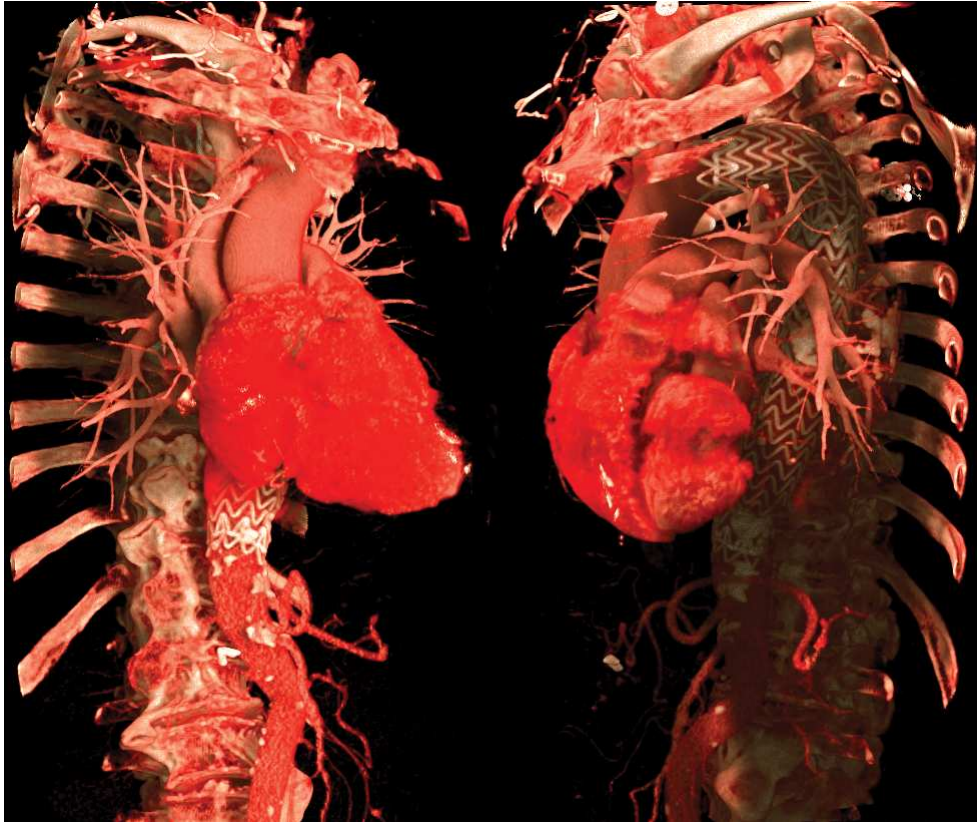


Figure 6.11: Two examples, with different positions of the light source, using the presented illumination technique for a medical volume (512^3 voxels) of a heart. A stent has been inserted into the aorta. ($I_{bias} = 0.1$, $R_{\Omega} = 16$ and $J = 32$)

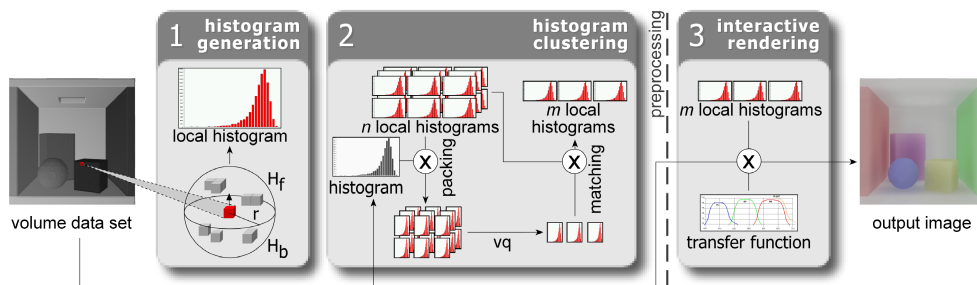
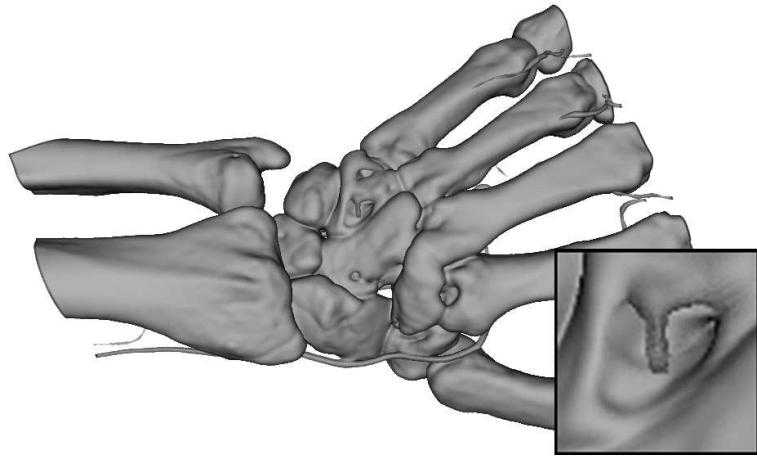
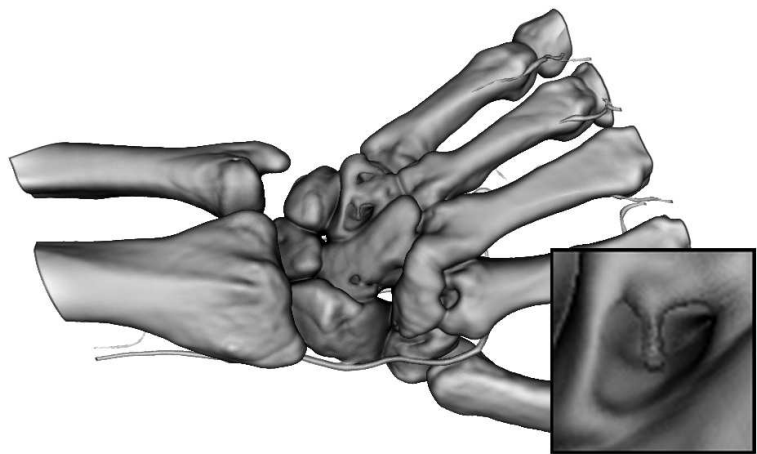


Figure 6.12: Workflow: In the first preprocessing stage a local histogram is generated for each of the n voxels in order to capture the distribution of intensities in its environment (1). Then the n local histograms are sorted into m clusters ($m < n$) through a vector quantization (vq). To accelerate the vector quantization, it operates on packed histograms. The packing is based on the histogram of the volumetric data set. After the clustering is finished the packed local histograms are replaced by their unpacked counterparts during the matching, before computing new cluster representatives (2). During rendering the local histograms representing the clusters are modulated with the transfer function to determine an environmental color for each voxel (3).



(a) Blinn-Phong



(b) our technique

Figure 6.13: A hand data set ($244 \times 124 \times 257$ voxel) rendered using our surface shading technique ($r=24$, $n_c=2048$) in comparison to Blinn-Phong shading. Notice the shading differences in the obscured areas.

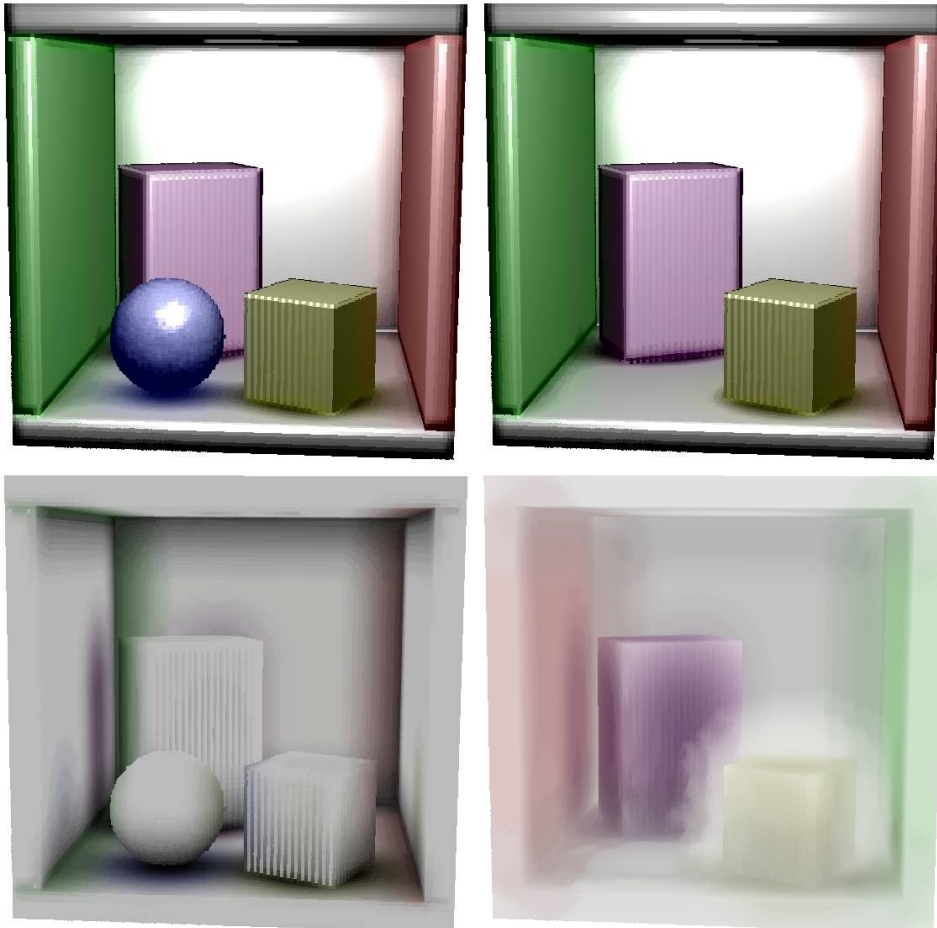
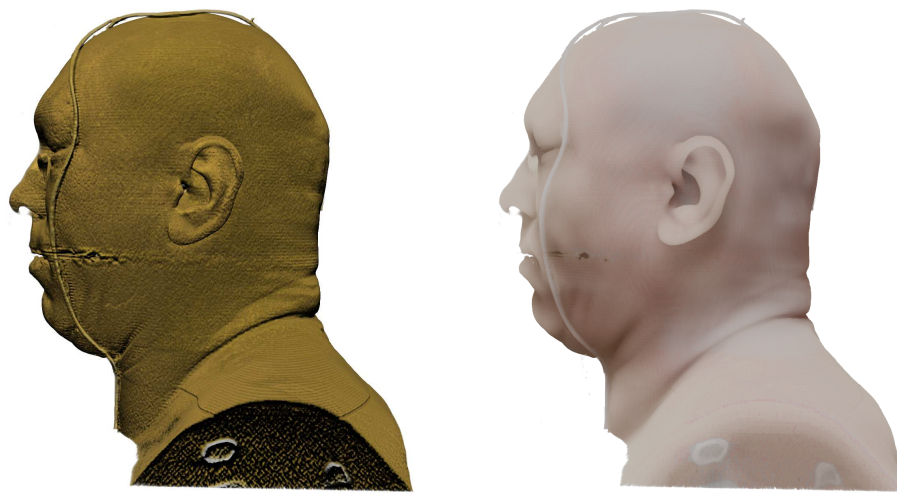


Figure 6.14: All four images show the same Cornell box data set ($r=32$, $n_c=1024$). For the right column the transfer function has been changed in such a way that the blue sphere disappears. The lower row shows the images with diffuse interreflections only (left) and material parameters set to simulate highly diffuse surfaces and an additional glow effect (right). The images have been rendered interactively by changing the transfer function resp. the glow mapping. Notice the color bleeding on the objects, which disappears for the blue sphere when removing it by modifying the transfer function.



(a) Blinn-Phong

(b) our technique

Figure 6.15: Our interactive volume rendering method ($r=20$, $n_c=2048$) applied to the Visible Human head data set ($192 \times 192 \times 110$ voxel). Both hemispheres, in direction of the gradient and the opposite direction, are considered during rendering, and thus in contrast to Blinn-Phong subsurface scattering effects as well as ambient occlusion in the inner parts of the auricle.

Course Notes

Advanced Illumination Techniques for GPU Volume Raycasting

Volume Scattering

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Patric Ljung

Siemens Corporate Research, Princeton, USA

Christof Rezk Salama

University of Siegen, Germany

Timo Ropinski

University of Münster, Germany



SIGGRAPH2009

Scattering Effects

7.1 Physical Background

Before we examine how to model realistic light propagation in transparent and translucent media, let us start by looking at the physical theory of light transport. In physics, light has been described by three different models:

Geometric Optics: Geometric optics consider rays of light. Rays are perpendicular to the wavefronts of the actual optical waves. They can be thought of as the paths of light particles (photons). Photons will get reflected or refracted at the interface between two media with different refractive index. Geometric optics is the usual way to describe light transport in computer graphics. It fails, however, to account for some optical effects such as diffraction and polarization.

Wave Optics: Wave optics consider light as electromagnetic radiation, as described by Maxwell's Equations. This model calculates the amplitude and phase of an electromagnetic wave as it passes through optical systems. It can account for diffraction, interference, and polarization effects, as well as aberrations and other complex effects. In computer graphics this model is rarely used mainly because the benefits hardly outweigh its computational complexity.

Quantum Optics: Quantum optics is the fundamental theory of light transport, which accounts for both particle and wave characteristics of light. It is the only explanation for the *photoelectric effect* as noted by Albert Einstein.

Although computer graphics mainly deals with geometric optics, there are many influences from the other models. The Monte-Carlo integration schemes described in this chapter for example may be considered a technique which accounts for the probabilistic characteristics of the motions of photons as derived by quantum mechanics.

7.2 Scattering

For surface rendering, light transport is usually assumed to take place in the vacuum. The interaction between light and matter is restricted to the surfaces of objects. In the vacuum, light travels unimpededly along straight lines (rays). Photons are reflected at surfaces, or refracted at the interface between dissimilar optical media. *Scattering* is the physical process which forces light to deviate from its straight trajectory. The reflection of light at a surface point is thus a scattering event. Depending on the material properties of the surface, incident photons are scattered in different directions and thereby change their energy and frequency, which possibly causes a change in color. For rough surfaces the direction in which individual photons are scattered vary within a considerably higher range (diffuse or *Lambertian* reflection) compared with shiny surfaces (specular reflection). This is the reason why we observe material types with a different appearance.

7.3 Single Scattering

To explain how scattering events are modelled in computer graphics, let's start with something familiar. Simple surface illumination may be achieved in computer graphics using single scattering events. Here,

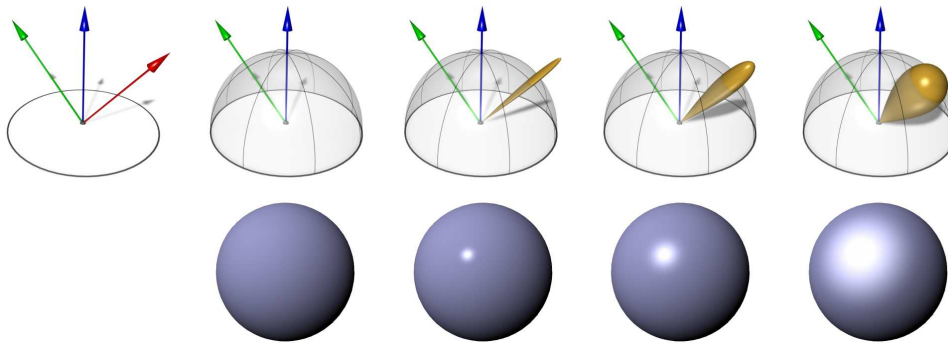


Figure 7.1: Simple single scattering as modelled by the Phong local illumination model with a single point light source. *From left to right:* In a *perfect mirror*, incident light (green arrow) is reflected about the surface normal (blue arrow) into the direction of perfect reflection (red arrow). With *Lambertian* reflection, incident light is scattered equally in all directions. With *specular* reflection light is scattered inside a *specular lobe* around the direction of perfect reflection.

a photon coming from a light source is scattered only once before it reaches the eye. In terms of ray-casting we may say that the ray changes its direction only once. Single scattering is closely related to *local illumination*. The only difference is that local illumination does *not* include shadows, while *single scattering* may account for occluded light sources as well.

In Figure 7.1, single scattering events are modelled using the well-known Phong illumination model. For the diffuse illumination term incident light is scattered equally across the entire hemisphere centered about the surface normal. For the specular term, scattering is restricted to a specular lobe centered about the direction of perfect reflection. More complex material properties may be modelled using the *bidirectional reflectance distribution function (BRDF)*.

The BRDF $f_r(\mathbf{x}, \lambda, \omega_i, \omega_o)$ at a single surface point \mathbf{x} is a function that defines how light is reflected at an opaque surface. The function takes an incoming light direction, ω_i , an outgoing direction, ω_o and a wavelength λ and returns the ratio of reflected radiance exiting along ω_o to the differential irradiance incident from direction ω_i . The wavelength parameter λ is usually omitted. In this case, the BRDF returns a chromatic RGB value $\hat{f}_r(\mathbf{x}, \omega_i, \omega_o)$.

In many applications, and especially real-time applications, the local illumination model is used in combination with point light sources, i.e. light is coming from exactly one or a few discrete directions. Local illumination with a BRDF and a single point light source can be formulated as a rendering equation:

$$L(\mathbf{x}, \omega_o) = \hat{f}_r(\mathbf{x}, \omega_i, \omega_o) \cos \theta_i \frac{L_i}{\pi r^2}, \quad (7.1)$$

where $L(\omega_o)$ is the radiance travelling from the surface point into direction ω_o . L_i is the emitted radiance of the light source, r is its distance from the surface point, and θ_i is the angle between the surface normal and the incoming direction ω_i .

In a more realistic scenario, however, incident light may come from all directions on the positive hemisphere Ω^+ centered about the surface normal. The observed radiance $L(\omega_o)$ must thus be calculated by integrating the incoming radiance from all directions ω_i over the hemisphere:

$$L(\mathbf{x}, \omega_o) = \int_{\Omega^+} \hat{f}_r(\mathbf{x}, \omega_i, \omega_o) \cos \theta_i L(\mathbf{x}, \omega_i) d\omega_i. \quad (7.2)$$

In mathematical terms, the rendering equation is a Fredholm equation of the second kind, which in general cannot be solved analytically. Nu-

merical solutions exist only if the BRDF is conservative, i.e.

$$\forall \omega_o, \mathbf{x}, : \int_{\omega^+} |\hat{f}_r(\mathbf{x}, \omega_i, \omega_o)| d\omega_i \leq 1. \quad (7.3)$$

To be physically plausible, BRDFs must also be positive definite and reciprocal. In chapter 8 we will see how the integral can be solved numerically using Monte-Carlo integration techniques. The BRDF notation can also be modified to account for transparent materials.

7.4 Indirect Illumination and Multiple Scattering

In computer graphics, single scattering accounts for light emitted from a light source directly onto a surface and reflected unimpededly into the observer's eye. Incident light $L(\omega_i)$ in Equation 7.3 thus comes directly from a light source. To generate more realistic images, we must account for both *indirect light* and *multiple scattering events*. Both concepts are closely related, they only differ in their scale. Indirect illumination means that light is reflected multiple times by different objects in the scene. Multiple scattering refers to the probabilistic characteristics of a scattering event caused by a photon being reflected multiple times within an object.

7.4.1 Indirect Light

The traditional (deterministic) ray tracing technique as shown in Figure 7.2, *left* only accounts for a very special type of indirect illumination: Perfect mirror reflections and perfect refractions in transparent objects. Ray tracing assumes that the path of a photon after emission from a light source is deterministic. The ray is traced by accounting for mirror reflections and refraction until it hits a point on a surface with diffuse or glossy characteristics. At this point a local BRDF model (such as Blinn, Phong, etc) is evaluated.

In comparison, Figure 7.2, *right* shows a more realistic illumination scenario, which accounts for indirect illumination. In the left image the ceiling is completely black. If you look at the ceiling you will notice that the left part is rendered slightly red and the right part slightly blue. This is because photons that reach the left part of the ceiling were most likely reflected from the red wall. This typical indirect illumination

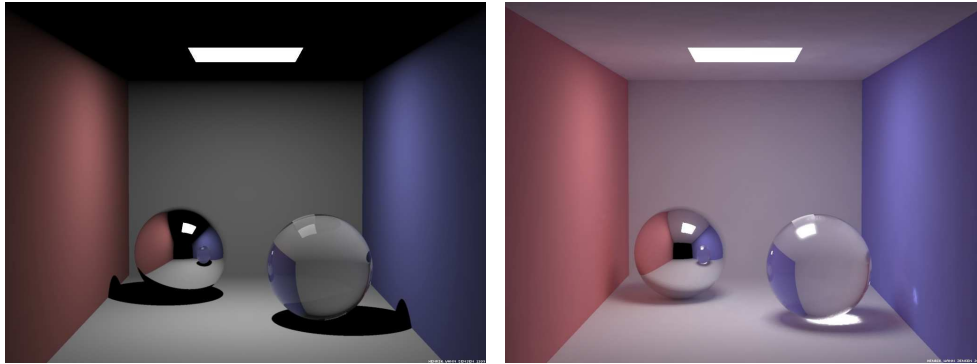


Figure 7.2: Ray tracing (left) and path tracing (right). Images courtesy of Henrik Wann Jensen, University of California, San Diego, USA

effect is called color bleeding. Another effect of indirect illumination is the caustic, the focussed light pattern caused by the transparent sphere.

7.4.2 Transparency and Translucency

Transparent materials are clear, so you can see through them. In contrast, translucent materials (such as frosted glass, clouds, milk etc.) allow light to pass through them only diffusely. Since ray-tracing relies on deterministic scattering events, it cannot account for translucent media, which requires the consideration of the probabilistic path of photons. The ray tracing technique thus neglects many possible paths that light can take from the light source to the eye. In general, the same is true for GPU-raycasting of volume data. We assume that the light travels along straight lines.

Figure 7.3 illustrates the difference between transparent and translucent material. The figure shows a CT scan of the UTCT Giant Salamander head rendered with different tissue properties. The internal structure of a transparent object are clearly visible, possibly distorted due to refraction (*left*). Depending on the optical properties (phase function), light is scattered in translucent materials (*middle and right*). In general, the exit point of a single photon is non-deterministic in translucent objects. In Chapter 9, we will see how such images can be generated using Monte-Carlo integration techniques which account for the probabilistic nature of light.



Figure 7.3: Volume Rendering example. Viewing rays penetrate transparent materials (*left*) almost unimpededly (disregarding refraction). Depending on the material properties, translucent objects let light pass through them more diffusely (*middle*, *right*). Transparency/translucency of the first iso-surface is controlled by a Fresnel term.

7.4.3 Phase Functions

Up until now, we have only talked about scattering events at surfaces, which can be described by a BRDF (Equation 7.3). Inside of translucent objects and participating media, scattering events are considered to potentially happen at every point inside this object. In this case, the BRDF f_r is replaced by the phase function h , and incoming radiance is integrated over the entire sphere Ω :

$$L_o(\mathbf{x}, \omega_o) = \int_{\Omega} h(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) d\omega_i. \quad (7.4)$$

The phase function describes the scattering characteristics of the participating medium. Note that the cosine term from Equation 7.3 is omitted in Equation 7.4, since the phase function directly operates on radiance values rather than differential irradiance like the BRDF.

The most popular phase function models are Henyey-Greenstein, Schlick, Mie and Rayleigh (see [41, 25, 81]).

The Henyey-Greenstein phase function is a simplified model for radiation in the galaxy. It describes the probability of a scattering events which changes the direction of a photon by a given angle. Compared with other models, the Henyey-Greenstein model does not incorporate any wavelength dependency. Nevertheless, it may be used to realistically model the optical properties of many natural phenomena, such as fog and clouds.

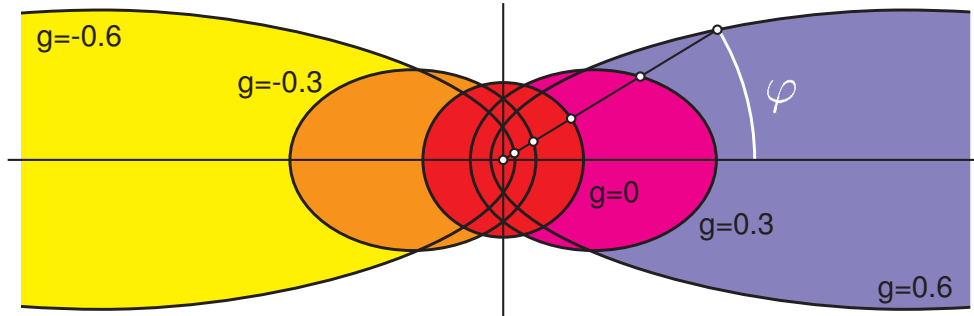


Figure 7.4: The Henyey-Greenstein phase function plotted for different anisotropy parameters g .

The probability of a photon changing its direction of motion by an angle of φ is approximated by the Henyey-Greenstein model as

$$G(\varphi, g) = \frac{1 - g^2}{(1 + g^2 - 2g \cos \varphi)^{\frac{3}{2}}}. \quad (7.5)$$

The parameter $g \in [-1, 1]$ describes the anisotropy of the scattering event. A value $g = 0$ denotes that light is scattered equally in all directions. A positive value of g increases the probability of forward scattering. Accordingly, with a negative value backward scattering will become more likely. If $g = 1$, a photon will always pass through the point unaffectedly. If $g = -1$ it will deterministically be reflected into the direction it came from. Figure 7.4 shows the Henyey Greenstein phase function for different anisotropy parameters g .

We can use the probability distribution from the Henyey-Greenstein model to construct a phase function according to Equation 7.4

$$h_{\text{HG}}(\mathbf{x}, \omega_i, \omega_o) = \frac{1 - g^2}{4\pi (1 + g^2 - 2g (\omega_i \cdot \omega_o))^{\frac{3}{2}}} \quad (7.6)$$

with $\cos \varphi = (\omega_i \cdot \omega_o)$.

7.4.4 Scattering at Transparent Surfaces

The phase function model is ideal for describing scattering events in participating media, where the probability of a scattering events depends only on the angle between the incoming and outgoing direction, regardless of their explicit orientation (rotation invariance). In contrast, the

BRDF at a given point depends on the orientation of the normal vector of the surface element. If we want to render images from volume data obtained by tomographic scans, however, we are not only interested in the scattering events in the homogenous regions, but also in the surfaces contained in the volume data. In this case we need a phased function which has an orientation, just like the normal vector in the BRDF.

An alternative to using the phase function here is to supplement the BRDF by a transmissive term. For semi-transparent surfaces, scattering events are still considered to happen only at surface boundaries, but light can be transmitted through transparent or translucent materials. The BRDF in Equation 7.3 is supplemented by a bidirectional transmittance distribution function (BTDF) \hat{f}_t defined on the opposite hemisphere. Both the BRDF and the BTDF are often considered together as a single bidirectional scattering distribution function (BSDF). The BSDF $\hat{f}(\mathbf{x}, \omega_i, \omega_o)$ leads to a rendering equation according to

$$L_o(\mathbf{x}, \omega_o) = \int_{\Omega} \hat{f}(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) |(\mathbf{n} \cdot \omega_i)| d\omega_i, \quad (7.7)$$

and can easily be converted to the phase function notation (Eq. 7.4) by incorporating the cosine term:

$$h_{\text{BSDF}}(\mathbf{x}, \omega_i, \omega_o) = \hat{f}(\mathbf{x}, \omega_i, \omega_o) |(\mathbf{n} \cdot \omega_i)|. \quad (7.8)$$

For volume rendering in practice, this means that we may account for surfaces inside the data, e.g. by looking at the gradient magnitude. If the gradient magnitude exceeds a given threshold ψ , we can use a BSDF to model the scattering events using the normal vector of the isosurface at that point (which coincides with the normalized gradient vector). In rather homogeneous regions, where the gradient magnitude is small, we model rotation-invariant scattering events according to the Henyey-Greenstein phase function. For a given scalar field $s(\mathbf{x})$, such as a tomographic scan of an object, a suitable phase function would thus be

$$h(\mathbf{x}, \omega_i, \omega_o) = \begin{cases} h_{\text{BSDF}}(\mathbf{x}, \omega_i, \omega_o) & \text{with } \mathbf{n} = \frac{\nabla s(\mathbf{x})}{|\nabla s(\mathbf{x})|}, \text{ if } |\nabla s(\mathbf{x})| > \psi \\ h_{\text{HG}}(\mathbf{x}, \omega_i, \omega_o), & \text{otherwise} \end{cases}$$

The free parameters of the model (e.g the anisotropy g in h_{HG}) may be obtained as a function of the scalar value $s(\mathbf{x})$ as well using a transfer function.

Now we have seen the equations we need to solve to build scattering into our GPU-based volume ray-caster. In the following chapter, we will see how we can solve these integrals using Monte-Carlo techniques.

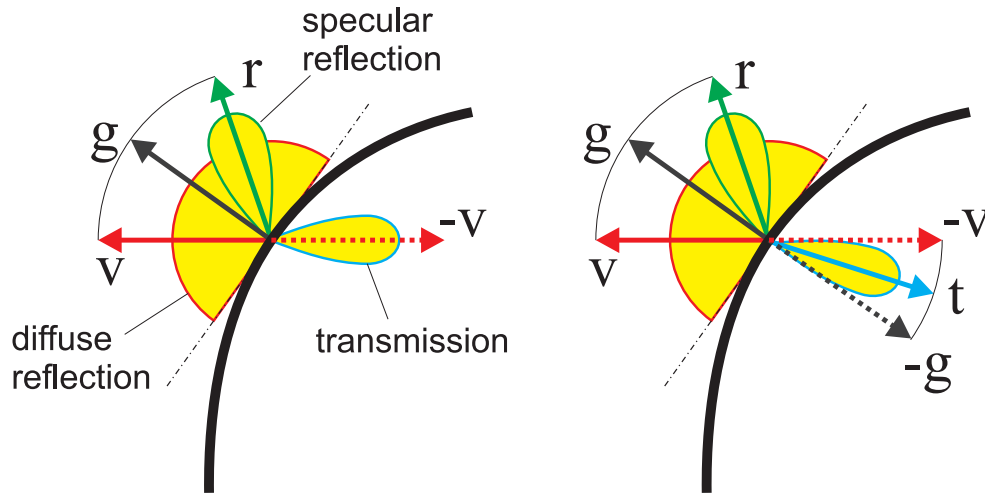


Figure 7.5: Illustration of the diffuse, specular and transmissive scattering component of our phenomenological phase function model without refraction *left* and with refraction *right*.

7.5 A Practical Phase Function Model

In an example in Chapter 9.3 we will use a simple phenomenological phase function model, which is equal to the BSDF at specified isosurfaces and otherwise contains a simple forward peak or a Henyey-Greenstein term as described above. The parameters of this phase function model are derived from the underlying scalar field $s(\mathbf{x})$. To keep the model controllable by the user, we restrict scattering events to happen at a fixed set of isosurfaces. Between these isosurfaces the ray direction does not change, but attenuation may still happen.

At the specified isosurfaces, the gradient magnitude $\nabla s(\mathbf{x})$ is guaranteed to be non-zero. The gradient vector is normalized and its orientation is adjusted to match the viewing direction. As in most illumination models we assume the viewing vector \mathbf{v} to point towards the eye position.

$$\mathbf{g}(\mathbf{x}) = \begin{cases} \frac{\nabla s(\mathbf{x})}{\|\nabla s(\mathbf{x})\|} & \text{if } \nabla s(\mathbf{x}) \cdot \mathbf{v} \geq 0 \\ -\frac{\nabla s(\mathbf{x})}{\|\nabla s(\mathbf{x})\|} & \text{if } \nabla s(\mathbf{x}) \cdot \mathbf{v} < 0 \end{cases} \quad (7.9)$$

Our phenomenological BSDF is illustrated in Figure 7.5. The reflective part f_r is equal to the specular and diffuse term of the Phong local

illumination model,

$$f_r = f_{\text{diff}} + f_{\text{spec}} \quad (7.10)$$

$$f_{\text{diff}}(\mathbf{v} \leftarrow \omega_i) = k_d (\mathbf{n} \cdot \omega_i) \quad (7.11)$$

$$f_{\text{spec}}(\mathbf{v} \leftarrow \omega_i) = k_s (\mathbf{r} \cdot \omega_i)^s \quad (7.12)$$

$$\text{with } \mathbf{r} = 2\mathbf{n} (\mathbf{n} \cdot \mathbf{v}) - \mathbf{v}. \quad (7.13)$$

The transmissive part scatters the transmitted light in an additional Phong lobe centered around the negative viewing vector $-\mathbf{v}$ in case of non-refractive transmission,

$$f_t(\mathbf{v} \leftarrow \omega_i) = k_t (-\mathbf{v} \cdot \omega_i)^q. \quad (7.14)$$

For refractive transmission, the Phong lobe is centered around the refracted ray direction \mathbf{t} (Figure 7.5, right). In this case the refracted vector \mathbf{t} is calculated according to Snell's law and replaces $-\mathbf{v}$ in Equation 7.14.

Figure 7.3 shows the influence of the exponent q for the transmission lobe.

The resulting BSDF model has 5 parameters to adjust for each specified isosurface: the diffuse, specular and transmissive material coefficients k_d , k_s and k_t and the exponents s and q for the specular and transmissive lobe.

7.6 Further Reading

Material properties of translucent surfaces, such as skin or paper, are often modeled using the bidirectional surface scattering reflectance distribution function (BSSRDF), which require two surface locations to be specified. A practical model has been proposed by Jensen et al. [46]. Donner et al. [22] have supplemented this model for multi-layered translucent materials. Interactive rendering technique for translucent surfaces have been presented by Lensch et al. [59] and Carr et al. [10].

Monte-Carlo Intergration

In the previous section we have seen that in order to generate physically plausible images, we need to solve Equation 7.4. In order to obtain the radiance at a given point \mathbf{x} in a given direction ω_o , we need to integrate the incident radiance over the sphere around the point, weighted by the phase function. I have also noted, that this integral cannot be solved analytically in general. We thus need to use numerical integration schemes.

8.1 Numerical Integration

The most popular numerical integration scheme is the Riemann sum. Let's have a look at a simple one-dimensional integral of a function $f(x)$,

$$I = \int_a^b f(x)dx. \quad (8.1)$$

To solve this integral numerically using a Riemann sum, we approximate the function $f(x)$ by a piecewise constant function and calculate the area of the rectangular blocks as shown in Figure 8.1. The heights of the blocks are obtained by sampling the function in n equidistant intervals $\Delta x = \frac{b-a}{n}$. An approximation to the integral is thus obtained by

$$I = \int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i) \frac{b-a}{n} \quad \text{with } x_i = i \cdot \frac{b-a}{n} \quad (8.2)$$

The approximation error of this sum depends on the number of samples n . If n approaches infinity, the approximation error will become zero.

8.1.1 Blind Monte-Carlo Integration

What happens if we modify the Riemann sum, such that the function is sampled at randomized locations, as shown in Figure 8.1 *right?*. Let

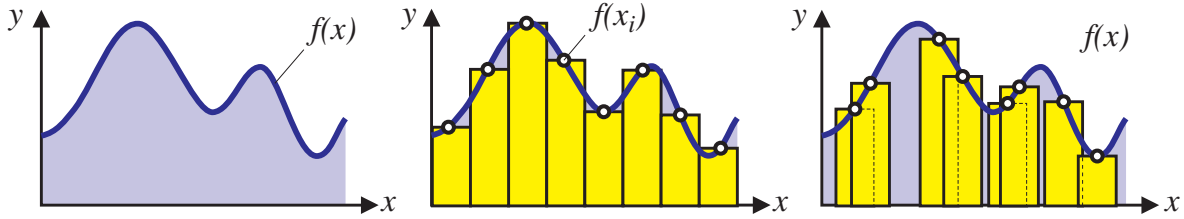


Figure 8.1: A function $f(x)$ (left) may be integrated by a Riemann sum, which approximates the function using piecewise constant blocks (middle). Alternatively, blind Monte-Carlo integration will sampling (right) samples the volume at random positions.

us assume that the samples are taken with a uniform probability distribution, which means that the probability of each sample position x_i is constant. We will use the same formula (Eq. 8.2) to estimate the integral, however, the sample position $x \in [a, b]$ now is a random variable with a uniform probability density $p(x)$:

$$\langle I \rangle = \sum_{i=0}^{n-1} f(x) \frac{b-a}{n} \quad \text{with } p(x) = \frac{1}{b-a} \quad (8.3)$$

To prove that such the estimator $\langle I \rangle$ is useful, we must show that it is unbiased, i.e that the expectation value of $\langle I \rangle$ is equal to the exact solution of the integral:

$$\begin{aligned} E[\langle I \rangle] &= E\left[\sum_{i=0}^{n-1} f(x) \frac{b-a}{n}\right] = \\ &= \frac{b-a}{n} \sum_{i=0}^{n-1} E[f(x)] = \\ &= \frac{b-a}{n} \sum_{i=0}^{n-1} \left(\int_a^b f(x)p(x) dx\right) = \\ &= \frac{b-a}{n} \sum_{i=0}^{n-1} \left(\int_a^b f(x) \frac{1}{b-a} dx\right) = \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \left(\int_a^b f(x) dx\right) = \\ &= \int_a^b f(x) dx = I \end{aligned} \quad (8.4)$$

We have now shown that the estimator $\langle I \rangle$ is unbiased and converges against the integral sought-after. This means that if the number of samples n approaches infinity, the estimation $\langle I \rangle$ will converge against the exact value of the integral, just like the Riemann sum. The practical meaning of this statement, however, is limited, because we have not yet investigated what happens if we use a finite number of samples n .

If you compare Figure 8.1 *middle* and *left*, your intuition is right, that the stochastic estimator is not as good as the Riemann sum. Indeed, one can show that for a growing number of samples n , the approximation error of the Riemann sum decreases much faster than the error of stochastic estimator $\langle I \rangle$. The Riemann sum converges *faster* to the exact solution.

The estimator $\langle I \rangle$ we have just seen is called a *blind Monte-Carlo estimator*. In the following section we will see that there are advantages of the blind Monte-Carlo estimator compared to the Riemann sum. The estimator is called *blind* because it does not make any assumptions about the function $f(x)$ to be integrated. In Section 8.3 we will also see how the convergence of the blind Monte-Carlo estimator can be improved by using a-priori information about the function $f(x)$

8.2 When Does Monte-Carlo Integration Make Sense?

In the previous section we have seen, that the blind Monte-Carlo estimator requires much more samples to achieve the same accuracy as the Riemann sum. Nevertheless, there are cases, especially in computer graphics, where the Monte-Carlo estimator is more efficient than the Riemann sum. The blind Monte-Carlo estimator has two important advantages:

- **Noise instead of aliasing:** If the function $f(x)$ contains high frequencies, such as the pattern in Figure 8.2 *top row*, equidistant sampling will inevitably lead to aliasing artifacts as can be seen in the right column of the figure. Even if the number of samples is increased by a factor of 16, aliasing will still be strongly visible (*bottom left*). Stochastic sampling may have a higher approximation error, nevertheless the images are visually more pleasing, since the human eye is less sensitive to noise than to aliasing artifacts.
- **Independence of any grid structure:** The second benefit of stochastic sampling in general is its independence of a grid structure. This is best explained using an example: Let us assume we

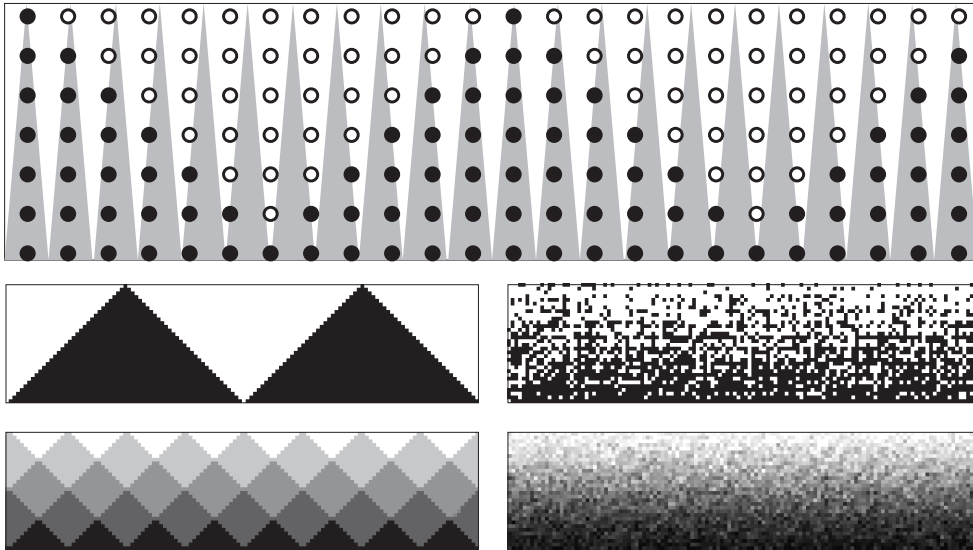


Figure 8.2: If a high-frequency pattern (*top row*) is sampled equidistantly below the Nyquist frequency, aliasing effects occur. The left column shows equidistant sampling, the right column stochastic sampling. The images in the middle row were generated with a single sample per pixel. The images in the bottom row were generated using 16 samples per pixel.

are going to integrate a three-dimensional function instead of the one-dimensional one in our example. We may try to integrate the function using a Riemann sum with $10 \times 10 \times 10 = 1000$ samples. Now if we find out that the approximation error is still too large, we need to increase the number of samples. The next possibility we have is to use $11 \times 10 \times 10$ samples, so we will have to evaluate 100 more samples in order to increase the accuracy. Since stochastic sampling does not require any sampling grid, we can increase the samples in steps of 1. Stochastic sampling is thus advantageous if the function to be integrated is high-dimensional and the evaluation of a sample is rather expensive.

In Figure 8.2 we have already seen the visual benefit of blind Monte-Carlo sampling. Stochastic sampling, however, may be further improved by integrating partial knowledge about the function to be integrated into the estimation process. We will discuss this in the following section which deals with importance sampling.

8.3 Importance Sampling

In our simple example in Section 8.1.1 we used a uniform probability density function (PDF) for determining the sample positions x (see Equation 8.3). Monte-Carlo integration, however, may be performed with an arbitrary PDF $p(x)$. The general Monte-Carlo integration formula is:

$$\langle I \rangle = \frac{1}{n} \sum_{i=0}^{n-1} \frac{f(x)}{p(x)} \quad (8.5)$$

We can easily show that this is a useful estimator again by calculating its expectation value:

$$\begin{aligned} E[\langle I \rangle] &= E\left[\frac{1}{n} \sum_{i=0}^{n-1} \frac{f(x)}{p(x)}\right] = \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \left(\int_a^b \frac{f(x)}{p(x)} p(x) dx \right) = \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \left(\int_a^b f(x) dx \right) = \\ &= \int_a^b f(x) dx = I \end{aligned} \quad (8.6)$$

Now, the question is: Can we find a PDF $p(x)$ which increases the convergence of the estimator $\langle I \rangle$? It turns out that the convergence of the Monte-Carlo estimator is optimal, if the probability density function $p(x)$ is a multiple of the function $f(x)$ itself. In practice this makes sense: We need to place more samples at positions x where the function $f(x)$ is large, and less samples where the function is small. This way we will focus the computational load to where the largest contribution to the integral is expected. However, we cannot use the function $f(x)$ directly as a PDF because it needs to be normalized:

$$p(x) = c \cdot f(x) \quad \text{and} \quad \int_a^b p(x) dx = 1 \quad (8.7)$$

This leads us to:

$$c = \frac{1}{\int_a^b f(x) dx} = \frac{1}{I} \quad (8.8)$$

You might notice the flaw: Calculating the constant c to normalize the ideal PDF would require us to know the sought-after integral I in ad-

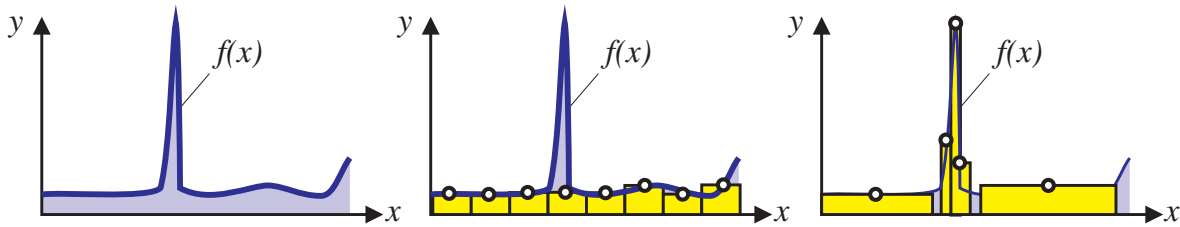


Figure 8.3: A function $f(x)$ with a sharp and narrow peak (*left*). Equidistant sampling of the Riemann sum might easily miss the peak. Based on a-priori information about the function $f(x)$, Monte-Carlo integration will place many samples at locations close to the peak.

vance. Nevertheless, even if we have only partial knowledge of the function to be integrated, we can significantly improve the convergence of the estimator.

Now, let us finally go back to our rendering equation:

$$L_o(\mathbf{x}, \omega_o) = \int_{\Omega} h(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) d\omega_i.$$

Although we do not know the integral completely, we have knowledge about the phase function $h(\mathbf{x}, \omega_i, \omega_o)$. We should thus shoot many rays into directions ω_i where the phase function h is high, and less samples where the phase function is low. Also, if we know anything about the lighting environment, we should shoot many rays into directions ω_i where $L(\omega_i)$ is expected to be large. This concept is known as *importance sampling* and significantly increases the convergence of the Monte-Carlo estimator.

Now you may see another important benefit of Monte-Carlo integration over the Riemann sum: Try to integrate a function $f(x)$ which has a known very sharp and narrow peak (Figure 8.3). Since Riemann integration does not take into account a-priori information about the function $f(x)$, its equidistant sampling might easily miss the narrow peak completely. If the location of the peak is known in advance, Monte-Carlo integration may place many samples at location near the peak and approximate the flat regions with fewer samples. With Riemann integration, the boxes in Figure 8.3 all have equal width. With Monte-Carlo estimation (see Eq. 8.5), the width w_i of a box in Figure 8.3, *right*, is depending on the probability of the sample:

$$w_i = \frac{1}{n \cdot p(x_i)} \quad (8.9)$$

This means, that the more likely a samples position x_i is, the more narrow is the box representing the sample.

8.4 GPU-based Importance Sampling

In this section we are having a look at different implementation of importance sampling on the GPU. In Chapter 9 we will see practical examples which apply these strategies for GPU-based volume ray-casting.

8.4.1 Focussing of Uniform Distribution

A fast and simple GPU-based technique for importance sampling, which is straight-forward to implement, has been used in [88]. Here, pre-computed random directions uniformly distributed on the unit sphere are used as basis. Simple but effective strategies to omit regions with only little contribution according to the phase function or BSDF are employed.

To avoid the necessity to account for different probability distributions within a shader program, this approach restricts itself to uniform distributions. In Section 8.4.2 we will see that the solid angle ω_i of a sample depends on its probability, just like the boxes in Figure 8.3 *right*.

Uniform samples are admittedly not the optimal sampling schemes, but they allow us to remove $p(x)$ from the sum in Equation 8.5 and replace the weighted sum by a simple average for efficiency. In Section 8.4.2 we will see a GPU-based implementation of importance sampling which deals with different pdfs in the shader.

8.4.1.1 Rejection Sampling

For a fast access to randomized direction vectors from within a fragment shader, a set of random value triplets representing points uniformly distributed on the unit sphere is pre-computed. We generate such vectors by the use of rejection sampling: We obtain triplets \mathbf{r}_S of uniformly distributed random values in the range of $[-1, 1]$. We discard all vectors with a magnitude larger than 1 and normalize the remaining vectors to unit length. The pre-computed random vectors are stored in a 2D and 3D texture. By sampling the texture at runtime, we can generate samples uniformly distributed on the unit sphere.

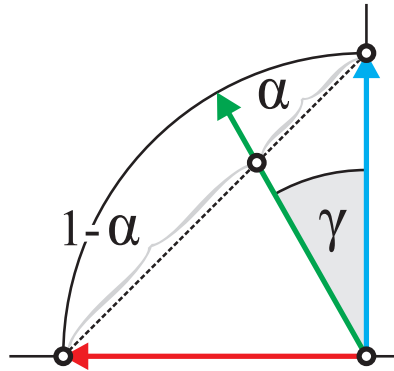


Figure 8.4: Geometric relationship between the interpolation weight α and the scattering cone angle γ .

8.4.1.2 Focussing

The random directions obtained from the texture can directly be used to sample the phase function. For diffuse, surface-like reflection, however, it is necessary to restrict the random directions to a hemisphere centered around a given unit vector \mathbf{n} . We can easily generate such samples by negating all random vectors outside the given hemisphere,

$$\mathbf{r}_H(\mathbf{n}) = \text{sgn}(\mathbf{n} \cdot \mathbf{r}_S)\mathbf{r}_S, \quad (8.10)$$

with sgn being the signum function.

For efficiently sampling a specular Phong lobe, we need to focus the sampling directions to a narrow cone centered around a given direction of reflection. A simple way of focussing ray directions is to compute a weighted sum of the hemispherical random samples \mathbf{r}_H and the direction of perfect reflection \mathbf{h} :

$$\begin{aligned} \tilde{\mathbf{r}}_P(\mathbf{h}) &= \alpha \cdot \mathbf{r}_H(\mathbf{h}) + (1 - \alpha)\mathbf{h}. \\ \mathbf{r}_P(\mathbf{h}) &= \frac{\tilde{\mathbf{r}}_P(\mathbf{h})}{\|\tilde{\mathbf{r}}_P(\mathbf{h})\|} \end{aligned} \quad (8.11)$$

The scalar weight α determines the maximum cone angle of scattering around the direction \mathbf{h} . A value $\alpha = 1$ means scattering in all directions on the hemisphere, while a value of $\alpha = 0$ results in the (non-randomized) ray direction perfectly focused into direction \mathbf{h} .

To determine an appropriate value of α for a given specular exponent s , we calculate the maximum reflection angle γ_{\max} , at which the specular

term falls below a user-specified threshold T (say 0.1),

$$\gamma_{\max}(s) = \max\{\gamma \mid \cos(\gamma)^s > T\}. \quad (8.12)$$

Solving this equation yields

$$\gamma_{\max} = \arccos(\sqrt[s]{T}). \quad (8.13)$$

Figure 8.4 illustrates the relationship between the focus weight α and the angle γ . The maximum angle between a hemispherical sample \mathbf{r}_H and the reflection direction \mathbf{h} is $\frac{\pi}{2}$. The interpolation according to Equation 8.11 moves the point along the dotted line and the normalization raises to interpolated point back to the hemisphere. From Figure 8.4, it is easy to derive a relationship between α and the maximum angle γ_{\max} by

$$\alpha = \frac{1 + \tan(\gamma_{\max} - \frac{\pi}{4})}{2} \quad (8.14)$$

The three sampling techniques outlined in this section should be sufficient to effectively increase the convergence of the Monte-Carlo estimator. Importance sampling requires knowledge about the scattering distribution at the surfaces. Which sampling strategy to use depends, of course, on the phase function model.

The drawback of this technique is that the threshold introduced in Equation 8.12 results in a Monte-Carlo estimator which is no longer unbiased. This means that the estimator will not converge against the correct solution of the integral. Instead a small error is introduced by truncating the specular lobes. Nevertheless, the described technique allows us to efficiently generate images at high visual quality, as will be shown in Chapter 9. A mathematically more accurate solution will be discussed in the following section.

8.4.2 Sampling of Reflection MIP-Maps

An efficient implementation of importance sampling has been published in volume 3 of the GPU Gems [14]. This implementation utilized area-preserving parameterizations of different BRDF models [81].

8.4.2.1 Area-Preserving Parameterizations

The Phong specular lobe centered around the reflection vector r , for example, may be written in polar coordinates ($\phi =$ azimuth, $\theta =$ elevation):

$$f_{\text{spec}}(\theta_i, \phi_i) = \cos^n \theta_i \sin \theta_i. \quad (8.15)$$

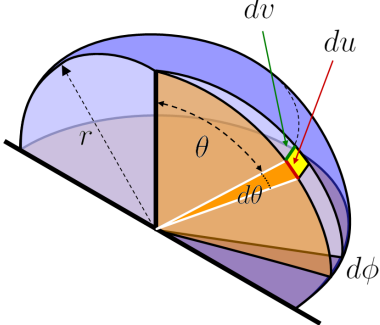
$$\begin{aligned}
 du &= r d\theta \\
 dv &= r \sin \theta d\phi \\
 dA &= du dv \\
 &= r^2 \sin \theta d\theta d\phi \\
 d\omega &= \frac{dA}{r^2} = \sin \theta d\theta d\phi
 \end{aligned}$$


Figure 8.5: The differential solid angle $d\omega$ of a surface element is the differential area dA projected onto the unit sphere. The area is calculated as width du times length dv , both of which are functions of the differential azimuth angle $d\phi$ and the differential elevation angle $d\theta$.

The additional sine term is introduced by changing from solid angle parameterization of the hemisphere to polar angle parameterization as outlined in Figure 8.5:

$$d\omega = \sin \theta d\theta d\phi \quad (8.16)$$

To calculate the probability density function for sampling the specular lobe, we need to normalize the lobe to integrate to one:

$$\begin{aligned}
 p(\theta_i, \phi_i) &= \frac{\cos^n \theta_i \sin \theta_i}{\int_0^{2\pi} \int_0^{\pi/2} \cos^n \theta_i \sin \theta_i d\theta d\phi} \\
 &= \frac{(n+1)}{2\pi} \cos^n \theta_i \sin \theta_i
 \end{aligned} \quad (8.17)$$

Note that this function is independent of the azimuth angle ϕ because of rotational symmetry. We can thus split the PDF into two independent equations for ϕ and θ :

$$p(\theta_i) = (n+1) \cos^n \theta_x \sin \theta_i \quad (8.18)$$

$$p(\phi_i) = \frac{1}{2\pi} \quad (8.19)$$

In practice, we can easily generate random variables $\psi \in [0, 1]$ with a uniform distribution (e.g. using the `rand()` function in C/C++). Note that Equation 8.19 is already a uniform distribution within the range

$[0, 2\pi]$. Now we need a mapping from a uniform distribution to the PDF in Equation 8.18. We calculate the inverse of the cumulative PDF:

$$\begin{aligned} P(\theta_i) &= \int_0^{\theta_i} p(\theta) d\theta \\ &= \cos^{(n+1)} \theta_i \end{aligned} \quad (8.20)$$

$$P^{-1}(\psi) = \arccos \left(\psi^{\left(\frac{1}{n+1}\right)} \right) \quad (8.21)$$

Now, if we have a pair of uniformly distributed random variables $(\psi_1, \psi_2) \in [0, 1] \times [0, 1]$, we can generate a random direction:

$$\theta_i = \arccos \left(\psi_1^{\left(\frac{1}{n+1}\right)} \right) \quad (8.22)$$

$$\phi_i = 2\pi \psi_2 \quad (8.23)$$

We can then convert the polar coordinates back to direction vectors in cartesian coordinates.

Since these samples are not uniformly distributed like in the focussing approach, we need to take into account the solid angle ω_i for each sample:

$$\omega_i = \frac{1}{n \cdot p(\theta_i, \phi_i)}. \quad (8.24)$$

Compared to the focussing approach, we will have to weight each sample by its corresponding solid angle: We divide each sample (θ_i, ϕ_i) by its probability $p(\theta_i, \phi_i)$ before averaging the samples.

8.4.2.2 Reflection MIP-Map

In the approach described in [14], a mip-mapped parabolic environment map is used to store the incident light averaged within different solid angles. The MIP-levels of the reflection map contain pre-filtered environment maps [40] for a pre-defined range of solid angles. The MIP-level for a texture sample is then calculated as a function of its solid angle (Eq. 8.24). This restricts the technique to local illumination of surfaces or isosurfaces. Furthermore, filtering of environment maps is restricted to BRDFs with a rotational symmetric specular lobe.

8.4.2.3 Filtering Environment Maps

The filtering of an environment map, however, can also be performed using Monte-Carlo integration. We can easily generate a filtered version

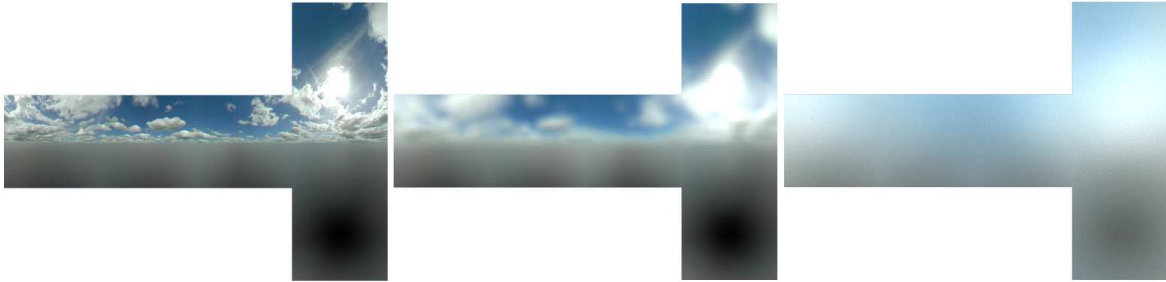


Figure 8.6: Examples of filtered environment maps: Original environment map (*left*). Filtered version with a specular exponent $s = 100$ (*middle*). Irradiance map created by filtering over the entire hemisphere (*right*). (Shader code in Listing 8.1)

of an environment map by rendering a screen-spaced quad (with the correct normal vectors specified at the vertices) into the six faces of a cube map (render-to-texture). The filtering is performed by the fragment shader shown in Listing 8.1. This shader uses the focussing technique from Section 8.4.1. For a given specular exponent s , the parameter `focus` must be calculated according to Equation 8.13. The shader reads from an arbitrary environment cube map and a pre-defined 3D texture containing unit random directions calculated using rejection sampling. Examples of filtered environment maps are shown in Figure 8.6.

8.5 Further Reading

In this chapter we have covered the theory of Monte-Carlo integration to an extent which is sufficient to understand the rest of the course. More information on Monte-Carlo integration can be found in the book by Pharr and Humphries[81], or in the SIGGRAPH 2002 course notes on "Advanced Global Illumination" by Philip Dutré, Kavita Bala and Philippe Bekaert.

```

1  #define NUMSAMPLES  (200.0)
2
3  float4 main(
4      float2 uv          : TEXCOORD0,
5      float3 normalIn   : NORMAL,
6      uniform float4    randSeed,
7      uniform float     focus,
8      uniform sampler3D noiseTex,
9      uniform samplerCUBE envMap
10 ) : COLOR
11 {
12     float3 N      = normalize(normalIn);
13     float4 sample = 0.0;
14
15     for(float i = 0; i < NUMSAMPLES; ++i) {
16
17         // calculate a randomized 3D texture coordinate
18         float offset = randSeed.a * i/NUMSAMPLES;
19         float3 randUV = (NormalIn + offset*randSeed.xyz);
20
21         // sample a precalculated 3D noise texture
22         // to obtain a randomized sampling direction
23         float3 randDir = expand(tex3D(noiseTex,randUV));
24         randDir = normalize(randDir);
25
26         // invert direction of back-facing
27         float cosTheta = dot(randDir,N);
28         if cosTheta < 0.0) {
29             randDir = -randDir;
30             cosTheta = -cosTheta ;
31         }
32
33         // focus the direction to specular lobe
34         randDir = normalize(lerp(randDir,N,focus));
35
36         // sample the environment cube
37         sample += cosTheta * texCUBE(envMap,randDir);
38     }
39     // average all samples
40     sample /= NUMSAMPLES;
41
42     return sample;
43
44 }

```

Listing 8.1: Cg fragment shader for filtering of environment maps. The shader is intended to read from an environment cube map and render into the faces of a reflection cube map.

GPU-Based Monte-Carlo Volume Raycasting

9.1 Monte-Carlo Techniques for Isosurfaces

In the first part of these course notes we have seen the basic implementation of GPU-based ray-casting. For each pixel it casts a ray into the volume and samples the volume on a straight line. If the volume is stored in a single 3D texture, however, we have the freedom to modify the ray direction to account for multiple scattering events with respect to a given phase function. We then of course need to cast multiple rays per pixel.

As an initial implementation, we may modify the fragment program for GPU-raycasting to calculate the first directional derivative of the scalar field along the viewing ray using central differences. Let $s(\mathbf{x})$ be the scalar field, and $x_i, i \in \{0, 1, \dots, m\}$ be the sample positions along the viewing ray ($i = 0$ is closest to the eye):

$$\frac{ds(\mathbf{x})}{d\mathbf{v}} = \nabla s(\mathbf{x}) \cdot \mathbf{v} \approx \frac{s(x_{i+1}) - s(x_{i-1}))}{x_{i+1} - x_{i-1}} \quad (9.1)$$

If the magnitude of the first derivative is larger than a specified threshold, we assume a surface scattering event according to the phase function model described in Section 7.5. We process the scattering event by calculating a randomized direction using importance sampling and scatter the ray into this direction. The user-specified threshold restricts strong directional changes to isosurfaces with a high gradient magnitude, while rays may pass forward through rather homogenous regions of the volume. We restart the fragment program for GPU-based raycasting multiple times with different random values. The program samples the volume at equidistant positions along the ray and integrate the phase function while the ray travels through the volume. When the viewing ray leaves the volume's bounding box, the incident radiance is sampled

from an environment cube map. The accumulated radiance RGB triplet is finally written into the frame buffer and averaged with the previous passes.

This straight forward implementation has the problem that many calculations, such as determination of the first scattering event, are performed again and again in successive passes. To improve this, we use a multi-pass rendering technique to reuse as much information as possible. The different rendering passes are described in the following sections.

- **First Hit Pass:** The first pass calculates the intersection of the viewing rays with the first isosurface. It is reused in subsequent passes to start viewing rays directly at the relevant position.
- **Local Illumination Pass:** The local illumination pass utilizes Monte-Carlo integration techniques to calculate surface shading with shift-variant BRDFs.
- **Ambient Occlusion Pass:** The ambient occlusion pass calculates self-shadowing of the isosurface.
- **Scattering Pass:** The scattering pass accounts for translucency and transparency.



Figure 9.1: UTCT Cheetah skull (512³, 16bit). Isosurface with Phong illumination and Ambient occlusion.

The first three passes together may be used to generate high-quality renditions of isosurfaces, such as the one displayed in Figure 9.1 at interactive frame rates. These passes will be explained in detail in the following Section. The scattering pass is more computationally expensive and will be described in Section 9.3. It may be previewed at interactive rates with a reduced quality (more noise due to fewer samples). Rendering the scattering pass in final quality will take up to a few seconds.

9.2 Isosurfaces with Shift-Variant or Anisotropic BRDFs

The process of rendering high-quality shaded isosurfaces can be divided into three different tasks:

1. Extraction of the isosurface. This includes the determination of position, normal direction and other surface properties obtained from the scalar field.
2. Local illumination of the surface. This includes evaluation of shift-variant BRDFs at the surface.
3. Shadow calculation. We will use Monte-Carlo integration to calculate ambient occlusion.

Each task is performed in a separate rendering pass as outlined in the following sections:

9.2.1 First Hit Pass

In a first rendering pass the front faces of the bounding box are rasterized. This first pass simultaneously renders into two floating-point off-screen buffers using multiple-render-targets. The basic fragment shader is shown in Listing 9.1. The shader contains a loop that successively samples the volume along the viewing ray. The user has specified the scalar values of an isosurface. While sampling the volume along the ray, we continuously check if the specified isosurfaces was intersected. (In the sample code, we assume that the scalar value of the first sample is below the isovalue) If the first isosurface is hit, the shader breaks out of the loop. The 3D texture coordinate of the intersection point is written to the first render target. The gradient magnitude is calculated, and the vector is then normalized. Gradient direction and magnitude are stored

```

1  uniform sampler3D VolumeTexture : TEXTURE0;
2  uniform float isoValue;
3  uniform float4x4 World2Tex;
4
5  struct COLOROUT {
6      float4 value0 : COLOR0;
7      float4 value1 : COLOR1;
8  };
9
10 COLOROUT main(float3 uvw           : TEXCOORD0,
11              float3 vecCameraPos   : TEXCOORD1,
12              float3 vecRayPos      : TEXCOORD2)
13 {
14     COLOROUT retval;
15
16     float4 firstHit = 0..xxxx;
17     float4 gradient = 0..xxxx;
18     // ray direction
19     float3 dirRay = normalize(vecRayPos-vecCameraPos);
20
21     for(int j = 0; j < MAX_NUM_SAMPLES; j++) {
22         // sample volume at ray position
23         float3 uvw = mul(World2Tex,float4(vecRayPos,1.0));
24         float sample = tex3D(VolumeTexture, uvw).r;
25         // check for hit with isosurface
26         if (sample > isoValue) {
27             // calculate gradient
28             float3 grad = getGradient(uvw);
29             gradient.a = length(grad);
30             gradient.rgb = normalize(grad);
31
32             firstHit.rgb = uvw;
33             break;
34         }
35         // check if ray exits the volume without intersection
36         if ((uvw.x < 0.0) || (uvw.y < 0.0) || (uvw.z < 0.0) ||
37             (uvw.x > 1.0) || (uvw.y > 1.0) || (uvw.z > 1.0)) {
38             break;
39         }
40         // proceed one step along the ray
41         vecRayPos += STEPSIZE*dirRay;
42     }
43
44     retval.value0 = firstHit;
45     retval.value1 = gradient;
46
47 }

```

Listing 9.1: A basic Cg fragment shader for determining the first hit with a specified isosurface. A possible implementation of the function `getGradient` in line 28 can be found in Listing 9.2.

```

1  float3 getGradient(float3 rayPos) {
2
3      float3 grad;
4      grad.x = tex3D(VolumeTexture,rayPos + float3(EPSILON,0.0,0.0)).r -
5              tex3D(VolumeTexture,rayPos - float3(EPSILON,0.0,0.0)).r;
6      grad.y = tex3D(VolumeTexture,rayPos + float3(0.0,EPSILON,0.0)).r -
7              tex3D(VolumeTexture,rayPos - float3(0.0,EPSILON,0.0)).r;
8      grad.z = tex3D(VolumeTexture,rayPos + float3(0.0,0.0,EPSILON)).r -
9              tex3D(VolumeTexture,rayPos - float3(0.0,0.0,EPSILON)).r;
10     frad /= (2.0*EPSILON);
11     return grad;
12 }

```

Listing 9.2: Cg shader function for estimating the gradient vector of the volume using central differences.

```

1  // interval bisection
2  float3 start = vecRayPos-STEPSSIZE*vecStep;
3  float3 end   = vecRayPos;
4
5  for(int b = 0; b < 10; ++b) {
6      float3 testPos = (start+end)/2.0;
7      float3 uvw = mul(World2Tex,float4(testPos,1.0));
8      if (tex3D(VolumeTexture, uvw).a < isoValue) {
9          start = testPos;
10     } else {
11         end = testPos;
12     }
13 }
14
15 vecRayPos = (start+end)/2.0;
16 uvw = mul(World2Tex,float4(vecRayPos,1.0));
17

```

Listing 9.3: Cg shader function for improving the accuracy of the isosurface detection using interval bisection.

as a floating-point RGBA quadruplet in the second render target and the fragment program terminates.

The gradient vector of the scalar field is always perpendicular to the isosurface. It may be estimated numerically using finite differences scheme. Listing 9.2 shows a working implementation. The function `getGradient` obtains six additional texture samples around the intersection point to estimate the gradient vector using central differences. For a detailed derivation of the gradient estimation scheme, see Chapter 5.3.1 in [25].

To improve accuracy of the isosurface detection, a few iterations of interval bisection may be performed to come closer to the exact intersection point, as suggested in [38]. If the first sample larger than the

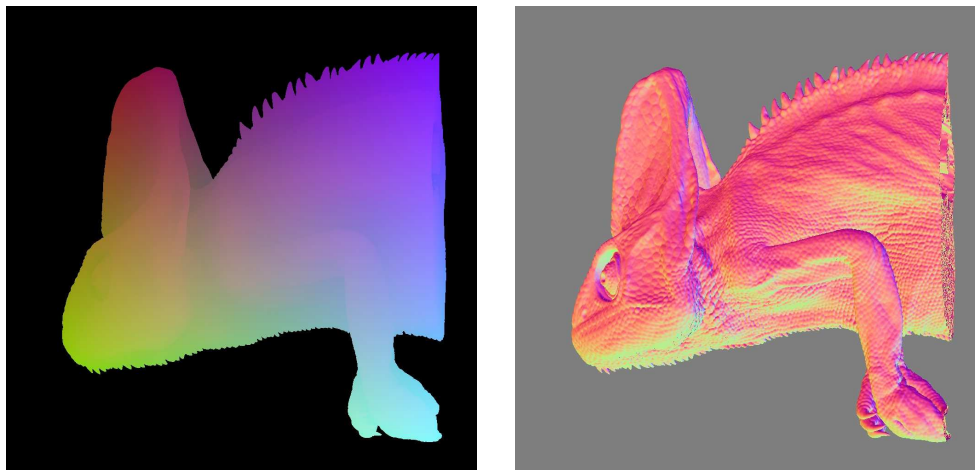


Figure 9.2: Results of the first hit rendering pass for the UTCT Veiled Chameleon data set. *Left:* texture coordinate of the first isosurface hit. *Right:* Gradient vector of the first isosurface.

isovalue is found, we go half a stepsize back and see if this sample is already larger than the isovalue. We proceed this way several times, halving the stepsize with each iteration. A code sample for interval bisection is shown in Listing 9.3. It may be inserted in Listing 9.1 right after line 21.

The contents of the two render targets for the first-hit pass are shown in Figure 9.2. Successive rendering passes start the ray integration directly at the intersection point with the first isosurface by reading the 3D texture coordinate determined in the first pass.

9.2.2 Deferred Shading Pass

Given the contents of the two rendering targets from the first hit pass, the surface shading for the isosurface may be calculated by reading the surface normal (the normalized gradient vector) from the second render target and evaluating a local illumination model, such as Phong with point light sources. If we want to account for environment light (e.g. stored in a cube map) we need to differentiate between shift-invariant BRDF whose reflectance behaviour can easily be precomputed, and shift-invariant BRDF which require Monte-Carlo integration to calculate the scattering events at run-time.

A *shift invariant* BRDF is a function $\hat{f}(\mathbf{x}, \omega_i, \omega_o)$ (see Equation 7.2) which does *not* depend on the position \mathbf{x} , i.e. the reflectance properties


```

1  #define NUMSAMPLES (50.0)
2
3  float4 main(
4      float2 screenPos : TEXCOORD0,
5      uniform float3   cameraPos,
6      uniform float4   randSeed,
7      uniform sampler2D firstHitPos,
8      uniform sampler2D firstHitGrad,
9      uniform samplerCUBE irradiancemap,
10     uniform samplerCUBE environmentMap,
11     uniform sampler3D  noiseTex
12 ) : COLOR
13 {
14     // read the information from previous pass
15     float4 rayPos = tex2D(firstHitPos, screenPos);
16     float4 gradient = tex2D(firstHitGrad, screenPos);
17     // focus parameter is the gradient magnitude in this example
18     float focus = gradient.w;
19
20     float3 V = normalize(rayPos.xyz-cameraPos);
21     float3 N = normalize(gradient);
22     float3 R = reflect(V,N);
23
24     float3 diffuse = texCUBE(irradiancemap, N);
25     float3 specular = 0.0;
26
27     for(float i = 0; i < NUMSAMPLES; ++i) {
28
29         // calculate a randomized 3D texture coordinate
30         float offset = randSeed.a * i/NUMSAMPLES;
31         float3 randUV = (rayPos + offset*randSeed.xyz);
32         // sample a precalculated 3D noise texture
33         // to obtain a randomized sampling direction
34         float3 randDir = expand(tex3D(noiseTex,randUV));
35         randDir = normalize(randDir);
36         // invert direction of back-facing
37         float cosTheta = dot(randDir,N);
38         if cosTheta < 0.0) {
39             randDir = -randDir;
40             cosTheta = -cosTheta ;
41         }
42         // focus the direction to specular lobe
43         randDir = normalize(lerp(randDir,R,focus));
44         // sample the environment cube
45         specular += cosTheta * texCUBE(environmentMap,randDir);
46     }
47     // average all samples
48     specular /= NUMSAMPLES;
49
50     return diffuse + specular;
51 }
52 }

```

Listing 9.4: Cg shader function for deferred shading using a shift-variant Phong BRDF. The diffuse term is sampled from a pre-filtered irradiance map. The specular term is calculated using stochastic sampling of an environment map

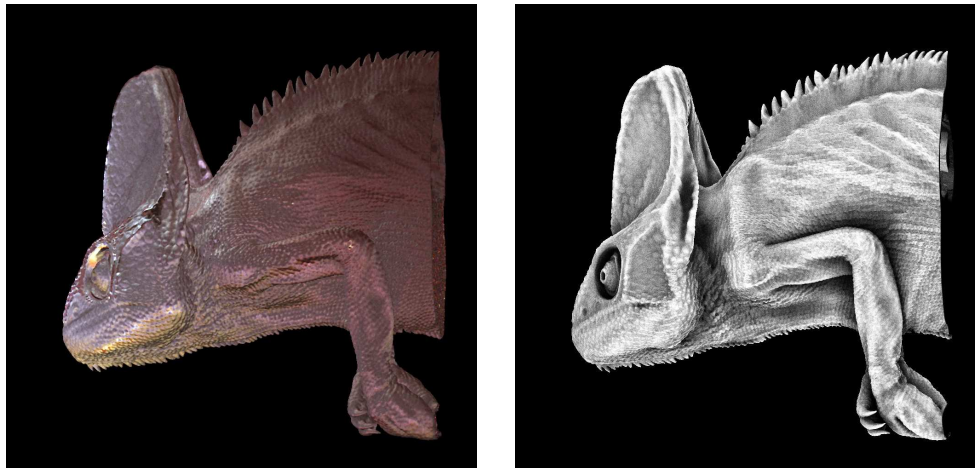


Figure 9.3: *Left*: Results of the deferred shading pass for the UTCT Veiled Chameleon data set. *Right*: Results of the deferred ambient occlusion pass.

are constant for the entire surface. To implement a shift-invariant BRDF, we can use an irradiance map for the diffuse term and a pre-filtered reflection map for the specular term (see Section 8.4.2.3), assuming that the specular lobe has rotational symmetry. For a shift-variant BRDF, the size of the specular lobe varies for different surface points \mathbf{x} . For a shift-invariant, but anisotropic BRDF, the shape of the specular lobe depends on the local orientation. In these cases, only the diffuse term may be pre-filtered, the specular term cannot be obtained by a single pre-filtered reflection map. We therefore modify Listing 8.1 to perform the Monte-Carlo integration directly on the surface. An example implementation is shown in Listing 9.4. The diffuse term is simply looked up in an irradiance cube map using the normal direction as texture coordinate. The specular exponent represented by the focus parameter (see Eq. 8.11) in this example is set to the gradient magnitude, which results in a shift-variant specular term. The Monte-Carlo integration is basically the same as in Listing 8.4.2.3. The only difference is that the specular lobe is focused onto the reflection vector \mathbf{R} instead of the normal \mathbf{N} . The result of the deferred shading pass is shown in Figure 9.3, *left*.

9.2.3 Deferred Ambient Occlusion Pass

Using the results from the first hit pass, we can as well calculate a deferred ambient occlusion pass for the isosurface. The fragment shader is

```

1  #define NUM_SAMPLES      (32.0)
2  #define FACTOR_RAYSTEP  (3.0)
3  #define MAX_NUM_RAY_STEPS (10.0)
4
5  float4 main(
6      float2 screenPos    : TEXCOORD0,
7      uniform float      isoValue,
8      uniform float4     randSeed,
9      uniform sampler2D  firstHitPos,
10     uniform sampler2D  firstHitGrad,
11     uniform sampler3D  volumeTex,
12     uniform sampler3D  noiseTex
13 ) : COLOR
14 {
15     // read the information from previous pass
16     float4 rayPos      = tex2D(firstHitPos, screenPos);
17     float4 gradient    = tex2D(firstHitGrad, screenPos);
18
19     if (gradient.w < EPSILON) return 0..xxxx;
20
21     float ambOcc = NUM_SAMPLES;
22
23     float3 pos;
24     float3 dir;
25     float3 dirStep;
26
27     for(float i = 0; i < NUM_SAMPLES; i+=1.0) {
28
29         // calculate a randomized sampling direction
30         float offset = randSeed.a * i/NUM_SAMPLES;
31         float3 randUVW = (rayPos + offset*randSeed.xyz);
32         float3 randDir = expand(tex3D(noiseTex,randUVW));
33         if (dot(randDir,N) < 0.0) randDir = -randDir;
34
35         // we do not really need to sample the full 180 degrees
36         randDir = normalize(randDir + 0.1*N);
37         // we sample with a larger stepsize for efficiency
38         dirStep = randDir * FACTOR_RAYSTEP * STEPSIZE;
39
40         // ensure the first sample point is outside
41         pos = rayPos + dirStep;
42
43         for(float s = 0; s < MAX_NUM_RAY_STEPS; s+=1.0) {
44             float value = tex3D(volumeTexture, pos).r;
45             if (value > isoValue) {
46                 ambOcc -= 1.0;
47                 break;
48             }
49             pos += dirStep;
50         }
51     }
52
53     return ambOcc.xxx;
54 }
55

```

Listing 9.5: Cg shader function for deferred isosurface ambient occlusion with Monte-Carlo sampling



Figure 9.4: UTCT Veiled Chameleon: Opaque isosurface of the skin rendered with a shift-variant BRDF. The specular exponent s proportional to the magnitude of the local gradient vector.

shown in Listing 9.5. In this example ambient occlusion is calculated in the local environment of the isosurface point. Starting from the surface point, random rays are shot across the hemisphere. For each ray, we check if the isosurface is hit again. The intensity value is the fraction of rays which do not hit the isosurface again. The result of the deferred ambient occlusion pass is shown in Figure 9.3, *right*.

The deferred shading pass and the ambient occlusion pass may be multiplied together for each pixel, resulting in a high-quality rendition of the isosurface. In the example image in Figure 9.4 showing the skin of the UTCT Veiled Chameleon CT scan, the shininess of the surface is proportional to the gradient magnitude at each surface point. In this data

set, high gradients of the isosurface are mainly caused by bone structures being close to the skin surface. If highest rendering performance is required, the number of samples for both the deferred shading pass and the ambient occlusion pass might be reduced, which still results in good quality images as shown in Figure 9.5.

9.3 Volume Scattering

In chapter 7.4 we have seen different phase function models which can be build into our GPU-based volume ray-caster. For volumetric scans, we want to differentiate between scattering at surfaces and scattering within rather homogeneous regions of the volume. This means that we approximate the directional derivative along the ray using finite differences and change the behaviour of the phase function accordingly.

While such an implementation is not very difficult, in practice, the visual appearance of this technique is hard to control by the user. We therefore suggest to directly specify a set of isovalues representing the isosurfaces at which the BSDF should be evaluated. Inbetween these isosurfaces a Henyey-Greenstein phase function or a simple forward peek with attenuation will be used.

Again, using the results of the first hit pass from the previous section, we now cast rays from the first isosurface into the interior of the object. The fragment shader for such a scattering pass will be basically the same as Listing 9.1. The only difference is that we start the rays directly at the first isosurface (reading the starting points from the first pass) and check for intersection of multiple iso surfaces simultaneously.

The code sample in Listing 9.6 shows a function which checks for multiple isosurface intersections. In this example we have specified four different isovalues $\hat{s}_i, i \in \{1, 2, 3, 4\}$. These values are written in ascending order ($\hat{s}_i < \hat{s}_{i+1}$) into the vector components of a `float4` value. The function `computeIsoIndex` takes a sample s and returns a `float4` value which contains a 1 in the respective vector component i , if the sample is inbetween \hat{s}_i and \hat{s}_{i+1} , and 0 otherwise. If the isovalues s_i for example are `isoValues = (0.1, 0.3, 0.5, 0.7)` and the sample value $s = 0.4$, this function will return a vector `isoIndex = (0.0, 1.0, 0.0, 0.0)`. To determine an isosurface intersection for successive samples s_i along the ray, we need to compare the `isoIndex` of the current sample to the previous sample. If the dot product between both `isoIndices` is zero, we have found an intersection with an isosurface. If there is no isosurface intersection we may proceed the sampling in one of the following ways:

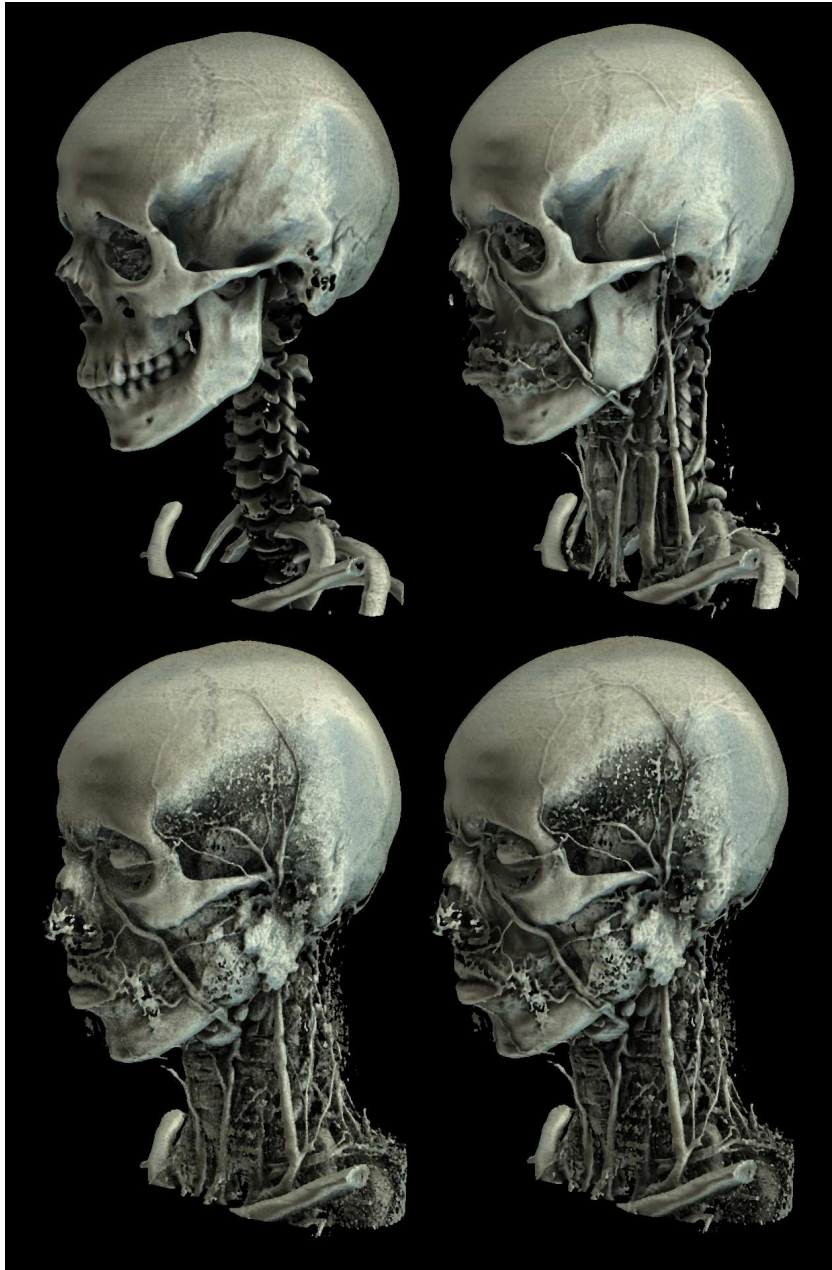


Figure 9.5: CT angiography of a human head. Different isovalue (*top row, bottom right*). The bottom left image was generated with a reduced number of samples for both shading and ambient occlusion. Although there is some noise visible in the image, the quality is still good.


```

1  float4 computeIsoIndex(float sample) {
2
3      float4 a;
4
5      a.x = (sample >= isoValues.x)? 1.0 : 0.0;
6      a.y = (sample >= isoValues.y)? 1.0 : 0.0;
7      a.z = (sample >= isoValues.z)? 1.0 : 0.0;
8      a.w = (sample >= isoValues.w)? 1.0 : 0.0;
9
10     return a - float4(a.yzw,0.0);
11 }
12 }

```

Listing 9.6: Cg shader function for calculating the isoIndex, which is used to check for intersection with four isosurfaces simultaneously.

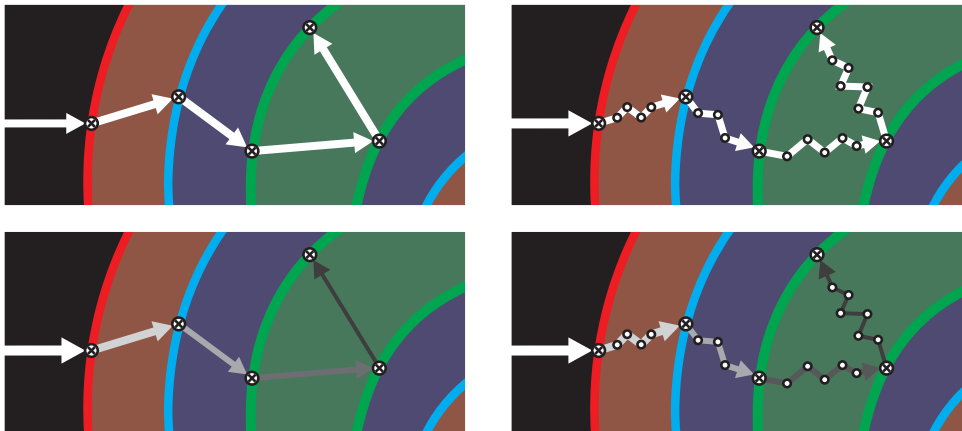


Figure 9.6: Different scattering effects inside the volume. *Left column:* Reflection at isosurfaces only. *Right column:* Reflection at isosurfaces and scattering in homogeneous regions. *Top row:* No attenuation. *Bottom row:* Attenuation of light due to absorption.

- process the next sample along the ray without changing the direction of the ray (no scattering). This refers to a Henyey-Greenstein phase function with an anisotropy parameter $g = 1$. (Figure 9.6, *top left*)
- process the next sample along the ray without changing the ray direction, but attenuate the radiance value. This refers to an absorption of light energy by the volume (Figure 9.6, *bottom left*) The radiance along the ray is multiplied by an attenuation factor $\tau \in [0, 1]$, similar to alpha blending. The attenuation factor may be either a constant (e.g. $\tau = 0.99$) or a function of the scalar field (implemented using a 1D texture lookup).
- scatter the next sample by changing the ray direction randomly. (Figure 9.6, *top right*) This refers to a Henyey-Greenstein phase function with an anisotropy value $g \in [0, 1[$. The lower the anisotropy value g , the slower the convergence will be. We will need a high number of samples to calculate isotropic scattering at every sample point. In practice it is thus useful to restrict the anisotropy to values close to 1.



Figure 9.7: An visual comparison of the different scattering implementations.

- combine both scattering and attenuation (Figure 9.6, *bottom right*).

We start a transmissive ray at the hit point with the first isosurface. The direction is scattered within a Phong lobe around the negative viewing direction or the refracted vector (see Figure 7.5). We trace this ray (with HG scattering and attenuation) until it hits the second isosurface. At this second hit point, the gradient vector is estimated and the ray is reflected randomly into the hemisphere centered around the gradient direction, and so on.

A visual comparison of the different scattering effects is shown in Equation 9.7. In this example, the optical properties were specified such that the internal surface structures of the volume still remain noticeable. Isosurface scattering alone will render in this example at about 0.3–0.5 seconds per frame in good quality. When reducing the number of samples such that the image becomes noise but quality is still acceptable, a frame rate of about 3 frames per second may be achieved.

If scattering is performed with every ray step, a random value must be fetched from the noise texture at every step which significantly increases the bandwidth load. The example images with scattering everywhere in the volume will render at about 1 seconds per frame in good quality. The difference between the different scattering effect in many cases is only marginal, so the benefit of implementing scattering events in homogeneous regions should be evaluated in any individual case.

9.3.1 Heuristic Simplifications

To improve the performance of the Monte-Carlo path tracing approach, we may apply some heuristics for simplification. I point out, however, that the techniques described in the following are not mathematically correct since they will lead to a bias in the Monte-Carlo estimator.

If we cast rays starting from the eye, the first few scattering events are the most important ones with respect to the final color. Scattering events which occur later along the path of the ray, are less important. With this heuristic in mind, we may simplify our implementation: We may decide to scatter the viewing ray 2 or 3 times at isosurfaces, and then sample the ray with attenuation only but without further scattering until it leaves the volume. This strategy avoids rays being reflected again and again until their contribution becomes zero due to attenuation.

Another simplification is based on the assumption that the attenuation which is accumulated from the eye point to the a hit point with an isosurface is an appropriate estimate for the attenuation from the

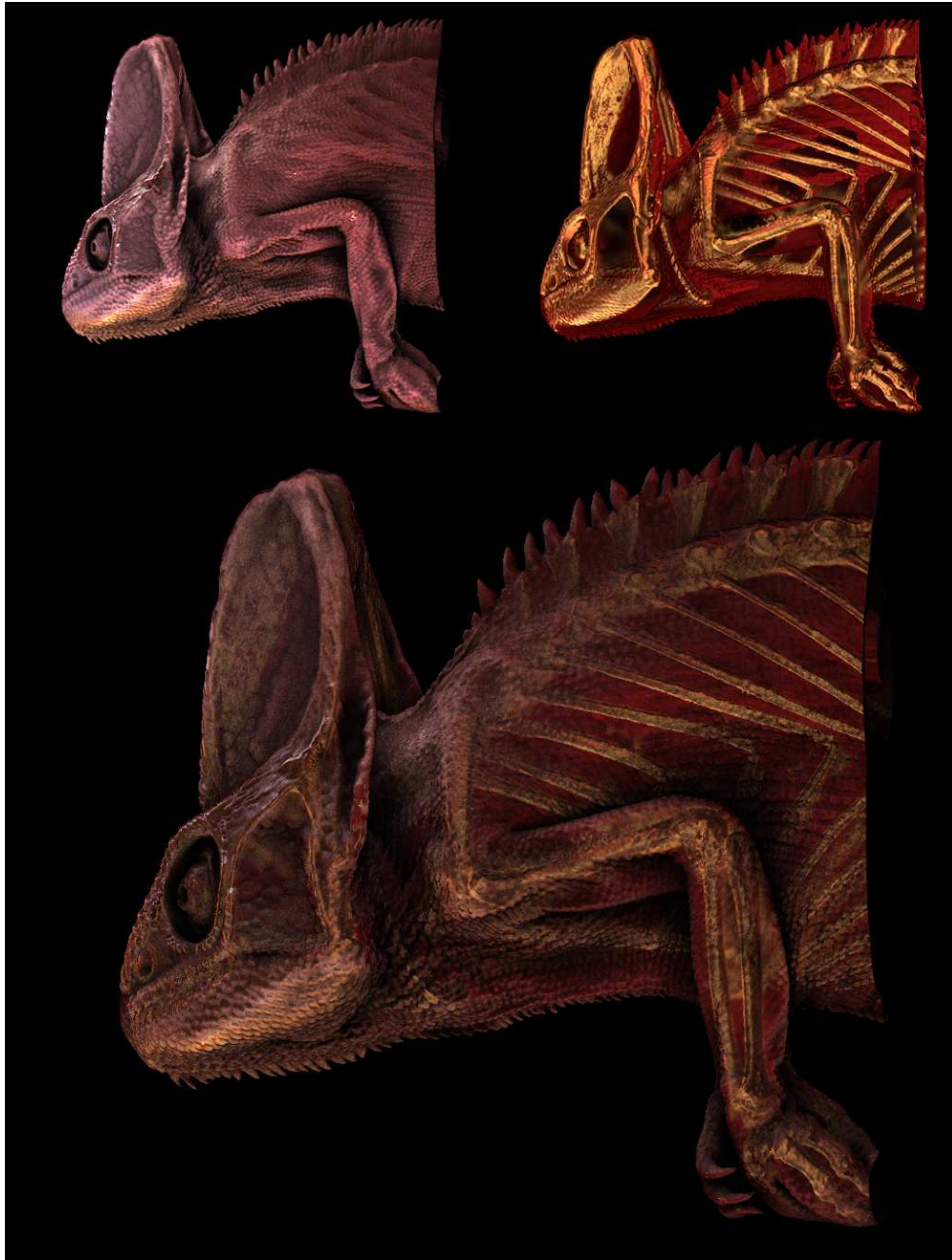


Figure 9.8: UTCT Chameleon data set. *Top left:* Isosurface with shading and ambient occlusion. *Top right:* Scattering pass. *Bottom:* Final composition.

hit point back to the outside. Hence, we can square the accumulated attenuation and directly sample the environment map in the reflected

direction without tracing the ray any further. Although less accurate, this technique is much faster because we can sample the environment map many times at the second hit point to directly estimate diffuse and specular reflection. An example image using this technique is displayed in Figure 9.9.

Our experiments have shown that there is little visible difference if we compare images generated by the more accurate and the faster method. We expect however, that this is due to the onion-like structure of iso-surfaces and should not be generalized to arbitrary surfaces inside the volume.



Figure 9.9: UTCT Big Brown Bat data set rendered with translucent skin.

Light Map Approaches

In combination with Shadow Volumes or Deep Shadow Maps (See chapter "shadows" in part 2 of these course notes), alternative implementations of scattering are possible. These approaches, however, are restricted to point light sources or directional lights.

A shadow volume is an additional voxel data set which contains information about the incident light at each voxel. It can be calculated slice-by-slice from a given point light source using a modified shadow map approach. The shadow volume covers the same space as the scalar field, and its slice planes are oriented perpendicular to the light direction. The resolution of the shadow volume may be lower than the original scalar volume. To generate a shadow volume, its slices are processed in front-to-back order from the viewpoint of the light source.

Each voxel of a slice to be processed is projected onto the previous slice (except for the first slice which uses the incident illumination of the light source directly). For point lights the position of the light source is the center of projection (perspective). For directional light sources the projection direction is the inverted light direction (parallel projection). Each slice voxel reads the incident light from the previous slice. The

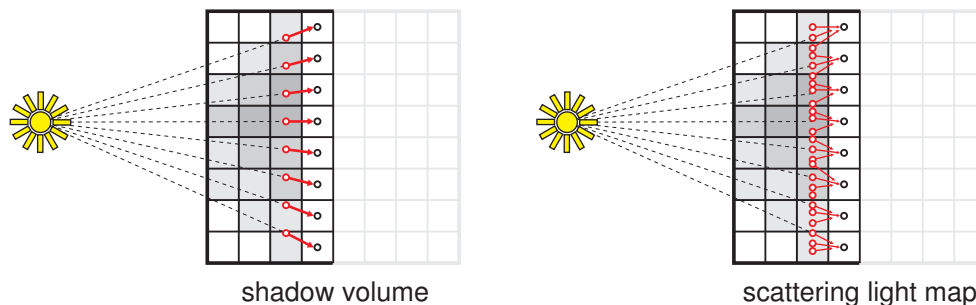


Figure 10.1: Shadow volume and 3D scattering light map. The shadow volume approach may be extended to sample the previous slice multiple times at locations scattered around the original sample position.

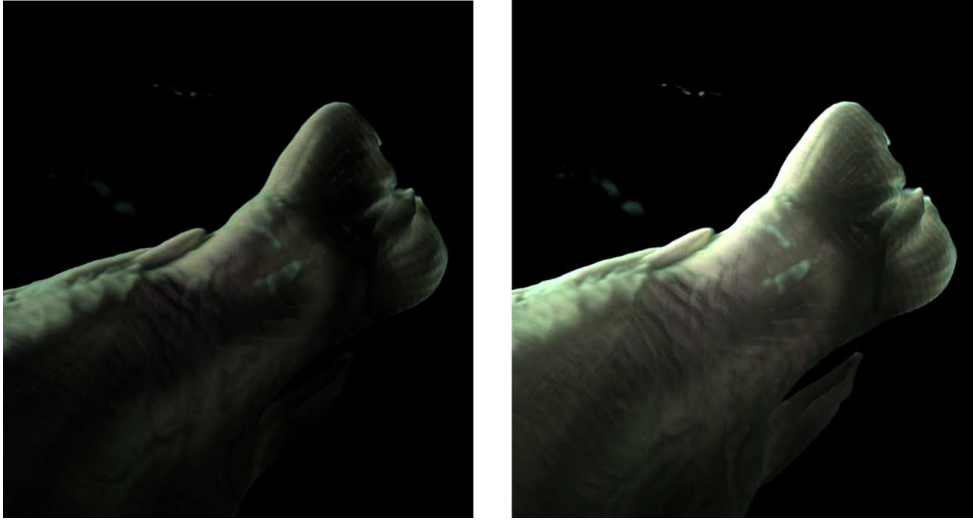


Figure 10.2: Tail fin of the carp data set. Shadow volume with direct light only (*left*) and 3D scattering light map with indirect light (*right*).

incident light is attenuated by the absorption of the scalar field generated by an opacity transfer function.

We can integrate scattering effects into the shadow volume during generation by sampling the previous slice multiple times per voxel, at locations randomly scattered around the original projected sample position. The distance of the random points from the original sampling location is controlled by a phase function model, such as Henyey-Greenstein. During raycasting the incident light from the shadow volume is added to the emission term during ray traversal. An example image of the shadow volume is shown in Figure 10.2.

A similar approach may be used with a deep shadow map instead of a shadow volume. In this case, the deep shadow map (which usually has a high resolution to account for fine details) is resampled on a lower uniform voxel grid. The low resolution for the light map is sufficient since illumination caused by multiple scattering effects is relatively low frequent. The resampled volume is then processed exactly the same way as the shadow volume. An example image of the combination of deep shadow map and low resolution light map is shown in Figure 10.3.

While being very efficient, there are some points which should be noted about the light map approaches:

- Like shadow volumes, scattering light maps are restricted to single

point light sources

- The light map needs to be recomputed whenever the position or direction of the light source changes
- The light map needs to be recomputed whenever the transfer function changes
- The approach for creating the light map as described above is more a soft shadow volume than a true scattering light map. In the algorithm described above and outlined in Figure 10.1, light is scattered only in slicing direction, so the scattering is most prominent for volumes lit from behind (forward scattering). For true subsurface scattering, more than one pass must be used to generate the 3D light map, to account for scattering events in all directions including backward scattering. Though this may be done easily, it will slow down the light map creation significantly.

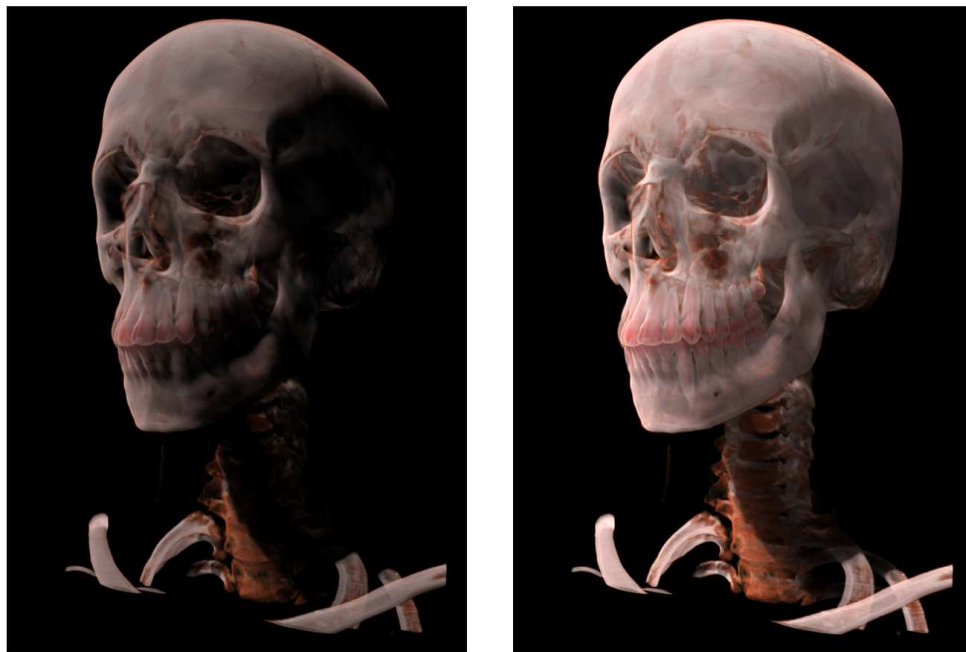


Figure 10.3: CT angiography data set. GPU-raycasting with a deep shadow map (*left*). High-resolution deep shadow map combined with a low-resolution scattering light map for indirect light (*right*).

Bibliography

- [1] Michael D. Adams. *The JPEG-2000 Still Image Compression Standard*. ISO/IEC (ITU-T SG8), September 2001. JTC 1/SC 29/WG 1: N 2412.
- [2] Chandrit Bajaj, Insung Ihm, and Sanghun Park. 3D RGB image compression for interactive applications. *ACM Transactions on Graphics*, 20(1):10–38, January 2001.
- [3] Kevin M. Beason, Josh Grant, David C. Banks, Brad Futch, and M. Yousuff Hussaini. Pre-computed illumination for isosurfaces. In *VDA '94: Proceedings of the conference on Visualization and Data Analysis '06 (SPIE Vol. 6060)*, pages 1–11, 2006.
- [4] Uwe Behrens and Ralf Ratering. Adding shadows to a texture-based volume renderer. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 39–46. ACM Press, 1998.
- [5] Johanna Beyer, Markus Hadwiger, Torsten Möller, and Laura Fritz. Smooth Mixed-Resolution GPU Volume Rendering. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 163–170, 2008.
- [6] Praveen Bhaniramka and Yves Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. In *Proceedings IEEE Visualization 2002*, pages 45–53, 2002.
- [7] J. F. Blinn. Jim blinn's corner: Image compositing–theory. *IEEE Computer Graphics and Applications*, 14(5), 1994.
- [8] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17:185–197, 2001.

- [9] A. R. Calderbank, Ingrid Daubechies, Wim Sweldens, and Boon-Lock Yeo. Wavelet transforms that map integers to integers. Technical report, Department of Mathematics, Princeton University, August 1996.
- [10] N. Carr, J. Hall, and J. Hart. GPU Algorithms for Radiosity and Subsurface Scattering. In *Proc. Graphics Hardware*, 2003.
- [11] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *HWWS '03: Proceedings of the conference on Graphics Hardware '03*, pages 51–59. Eurographics Association, 2003.
- [12] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization '98*, pages 167–174, 1998.
- [13] Yi-Jen Chiang, Cludio T. Silva, and Willam J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings IEEE Visualization 1998*, pages 167–174,530, 1998.
- [14] M. Colbert and J. Křivánek. *GPU Gems 3*, chapter GPU-Based Importance Sampling, pages 459–475. Addison-Wesley, 2007.
- [15] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings IEEE Visualization 1997*, pages 235–244, 1997.
- [16] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248. ACM Press, 1977.
- [17] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings SIGGRAPH '84*, volume 18, pages 207–212, 1984.
- [18] Carsten Dachsbacher and Marc Stamminger. Splatting indirect illumination. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 93–100, New York, NY, USA, 2006. ACM.
- [19] Ingrid Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.

- [20] Philippe Desgranges and Klaus Engel. US patent application 2007/0013696 A1: Fast ambient occlusion for direct volume rendering, 2007.
- [21] Philippe Desgranges, Klaus Engel, and Gianluca Paladini. Gradient-free shading: A new method for realistic interactive volume rendering. In *VMV '05: Proceedings of the international fall workshop on Vision, Modeling, and Visualization*, pages 209–216, 2005.
- [22] C. Donner and H. W. Jensen. Light Diffusion in Multi-Layered Translucent Materials. In *Proc. ACM SIGGRAPH*, 2005.
- [23] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proceedings of SIGGRAPH '88*, pages 65–74, 1988.
- [24] D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, July 1998.
- [25] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. AK Peters, 2006.
- [26] James D. Foley, Richard L. Phillips, John F. Hughes, Andries van Dam, and Steven K. Feiner. *Introduction to Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.
- [27] Jinzhu Gao, Jian Huang, C. Ryan Johnson, and Scott Atchley. Distributed data management for large volume visualization. In *Proceedings IEEE Visualization 2005*, pages 183–189. IEEE, 2005.
- [28] Jinzhu Gao, Jian Huang, Han-Wei Shen, and James Arthur Kohl. Visibility culling using plenoptic opacity functions for large volume visualization. In *Proceedings IEEE Visualization 2003*, pages 341–348. IEEE, 2003.
- [29] Jinzhu Gao, Han-Wei Shen, Jian Huang, and James Arthur Kohl. Visibility culling for time-varying volume rendering using temporal occlusion coherence. In *Proceedings IEEE Visualization 2004*, pages 147–154. IEEE, 2004.

- [30] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, 2004.
- [31] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Gröller. Memory efficient acceleration structures and techniques for CPU-based volume raycasting of large data. In *Proceedings IEEE Volume Visualization and Graphics Symposium*, pages 1–8, 2004.
- [32] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers and Graphics*, 28:719–729, 2004.
- [33] Stefan Guthe and Wolfgang Straßer. Real-time decompression and visualization of animated volume data. In *Proceedings IEEE Visualization 2001*, pages 349–356, 2001.
- [34] Stefan Guthe and Wolfgang Strasser. Advanced techniques for high quality multiresolution volume rendering. In *Computers & Graphics*, volume 28, pages 51–58. Elsevier Science, February 2004.
- [35] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Proceedings IEEE Visualization 2002*, pages 53–60, 2002.
- [36] Attila Gyulassy, Lars Linsen, and Bernd Hamann. Time- and space-efficient error calculation for multiresolution direct volume rendering. In *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*. Springer-Verlag, Heidelberg, Germany, 2006.
- [37] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proceedings of Eurographics 2005*, pages 303–312, 2005.
- [38] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In *Proceedings of Eurographics*, pages 303–312, 2005.

- [39] Markus Hadwiger, Andrea Kratz, Christian Sigg, and Katja Bühler. Gpu-accelerated deep shadow maps for direct volume rendering. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 49–52, New York, NY, USA, 2006. ACM Press.
- [40] W. Heidrich and H.-P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Proc. ACM SIGGRAPH*, 1999.
- [41] L. Henyey and J. Greenstein. Diffuse radiation in the galaxy. *Astrophysical Journal*, pages p. 70–83, 93.
- [42] Frida Hernell, Patric Ljung, and Anders Ynnerman. Efficient ambient and emissive tissue illumination using local occlusion in multiresolution volume rendering. In *Proceedings Eurographics/IEEE-VGTC Symposium on Volume Graphics*. Eurographics/IEEE, 2007.
- [43] Frida Hernell, Patric Ljung, and Anders Ynnerman. Interactive Global Light Propagation in Direct Volume Rendering using Local Piecewise Integration. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 105–112, 2008.
- [44] W. Hong, F. Qiu, and A. Kaufman. Gpu-based object-order raycasting for large datasets. In *Proceedings of Volume Graphics 2005*, 2005.
- [45] Insung Ihm and Sanghun Park. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999.
- [46] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A Practical Model for Subsurface Light Transport. In *Proceedings of ACM SIGGRAPH*, pages 511–518, 2001.
- [47] Ralf Kähler, John Wise, Tom Abel, and Hans-Christian Hege. Gpu-assisted raycasting for cosmological adaptive mesg refinement simulations. In *Proceedings Eurographics/IEEE Workshop on Volume Graphics 2006*, pages 103–110,144, 2006.
- [48] D. Kalra and A. H. Barr. Guaranteed ray intersections with implicit surfaces. In *Proceedings of SIGGRAPH '89*, pages 297 – 306, 1989.

- [49] A. Kaufman. Voxels as a Computational Representation of Geometry. In *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, 1994.
- [50] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182, London, UK, 2001. Springer-Verlag.
- [51] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.
- [52] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 109–116. IEEE Computer Society, 2002.
- [53] Joe Kniss, Simon Premoze, Charles Hansen, Peter Shirley, and Allen McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.
- [54] M. Kraus and T. Ertl. Adaptive texture maps. In *Proceedings of Graphics Hardware 2002*, pages 7–15, 2002.
- [55] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.
- [56] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings IEEE Visualization 1999*, pages 355–362, 1999.
- [57] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. Efficient error calculation for multiresolution texture-based volume visualization. In Gerald Farin, Bernd Hamann, and Hans Hagen, editors, *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 51–62. Springer-Verlag, Heidelberg, Germany, 2003.
- [58] Michael S. Langer and Heinrich H. Bühlhoff. Depth discrimination from shading under diffuse lighting. *Perception*, 29(6):649–660, 2000.

- [59] H. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. *Computer Graphics Forum*, 22(2), 2003.
- [60] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.
- [61] F. Link, M. Koenig, and H.-O. Peitgen. Multi-Resolution Volume Rendering with per Object Shading. In *Proceedings of Vision, Modeling and Visualization*, pages 185–191, 2006.
- [62] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2:73–84, 1996.
- [63] Patric Ljung. Adaptive sampling in single pass, GPU-based ray-casting of multiresolution volumes. In *Proceedings Eurographics/IEEE Workshop on Volume Graphics 2006*, pages 39–46,134, 2006.
- [64] Patric Ljung. *Efficient Methods for Direct Volume Rendering of Large Data Sets*. PhD thesis, Linköping University, Sweden, 2006. Linköping studies in science and technology. Dissertations no. 1043.
- [65] Patric Ljung, Claes Lundström, and Anders Ynnerman. Multiresolution interblock interpolation in direct volume rendering. In *Proceedings Eurographics/IEEE Symposium on Visualization 2006*, pages 259–266, 2006.
- [66] Patric Ljung, Claes Lundström, Anders Ynnerman, and Ken Museth. Transfer function based adaptive decompression for volume rendering of large medical data sets. In *Proceedings IEEE Volume Visualization and Graphics Symposium 2004*, pages 25–32, 2004.
- [67] Patric Ljung, Calle Winskog, Anders Perssson, Claes Lundström, and Anders Ynnerman. Full body virtual autopsies using a state-of-the-art volume rendering pipeline. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization/Information Visualization 2006)*, 12:869–876, 2006.

- [68] Tom Lokovic and Eric Veach. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.
- [69] Eric B. Lum, Kwan-Liu Ma, and John Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings IEEE Visualization 2001*, pages 263–270, 2001.
- [70] Eric B. Lum, Kwan-Liu Ma, and John Clyne. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics*, 8:286–298, 2002.
- [71] Claes Lundström, Patric Ljung, and Anders Ynnerman. Local histograms for design of transfer functions in direct volume rendering. *Transactions on Visualization and Computer Graphics*, 12(6):1570–1579, Nov.-Dec. 2006.
- [72] Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek. Interactive Volume Rendering with Ray Tracing. In *Eurographics State of the Art Reports*, 2006.
- [73] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.
- [74] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.
- [75] Jörg Mensmann, Timo Ropinski, and Klaus Hinrichs. Accelerating Volume Raycasting using Occlusion Frustum. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 147–154, 2008.
- [76] Ky Giang Nguyen and Dietmar Saupe. Rapid high quality compression of volume data for visualization. *Computer Graphics Forum*, 20(3), 2001.
- [77] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

- [78] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization '98*. IEEE-CS, ACM, October 1998.
- [79] A. Patra and M.D. Wang. Volumetric medical image compression and reconstruction for interactive visualization in surgical planning. In *Proceedings Data Compression Conference 2003*, page 442, March 2003.
- [80] Eric Penner and Ross Mitchell. Isosurface Ambient Occlusion and Soft Shadows with Filterable Occlusion Maps. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 57–64, 2008.
- [81] Matt Pharr and Greg Humphries. *Physically Based Rendering*. Morgan Kaufman, 2004.
- [82] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291. ACM Press, 1987.
- [83] S. Roettger, S. Guthe, D. Weiskopf, and T. Ertl. Smart hardware-accelerated volume rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238, 2003.
- [84] Timo Ropinski, Jens Kasten, and Klaus H. Hinrichs. Efficient Shadows for GPU-based Volume Raycasting. In *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization (WSCG08)*, pages 17–24, 2008.
- [85] Timo Ropinski, Jennis Meyer-Spradow, Stefan Diepenbrock, Jörg Mensmann, and Klaus H. Hinrichs. Interactive Volume Rendering with Dynamic Ambient Occlusion and Color Bleeding. *Computer Graphics Forum (Eurographics 2008)*, 27(2):567–576, 2008.
- [86] Stefan Röttger, Michael Bauer, and Marc Stamminger. Spatialized transfer functions. In *EuroVis*, pages 271–278, 2005.
- [87] Marc Ruiz, Imma Boada, Ivan Viola, Stefan Bruckner, Miquel Feixas, and Mateu Sbert. Obscure-based Volume Rendering Framework. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 113–120, 2008.

- [88] C. Rezk Salama. GPU-Based Monte-Carlo Volume Raycasting. In *Proc. Pacific Graphics*, 2007.
- [89] Mirko Sattler, Ralf Sarlette, Thomas Mücken, and Reinhard Klein. Exploitation of human shadow perception for fast shadow rendering. In *APGV '05: Proceedings of the 2nd symposium on Applied perception in graphics and visualization*, pages 131–134. ACM Press, 2005.
- [90] H. Scharsach, M. Hadwiger, A. Neubauer, S. Wolfsberger, and K. Bühler. Perspective Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications. In *Proceedings of Eurovis '06*, pages 315–323, 2006.
- [91] Henning Scharsach. Advanced GPU raycasting. In *Proceedings of the 9th Central European Seminar on Computer Graphics*, May 2005.
- [92] Jens Schneider and Rüdiger Westermann. Compression domain volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.
- [93] Peter-Pike Sloan, Jesse Hall, John Hart, and John Snyder. Clustered principal components for precomputed radiance transfer. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 382–391. ACM Press, 2003.
- [94] Peter-Pike Sloan, Ben Luna, and John Snyder. Local, deformable precomputed radiance transfer. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1216–1224. ACM Press, 2005.
- [95] Irwin Edward Sobel. *Camera models and machine perception*. PhD thesis, Stanford University, Stanford, CA, USA, 1970.
- [96] Lisa M. Sobierajski and Arie E. Kaufman. Volumetric ray tracing. In *VVS '94: Proceedings of the 1994 symposium on Volume Visualization '94*, pages 11–18. ACM Press, 1994.
- [97] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.
- [98] A. James Stewart. Vicinity shading for enhanced perception of volumetric data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 47. IEEE Computer Society, 2003.

- [99] Wim Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Journal of Applied and Computational Harmonic Analysis*, (3):186–200, 1996.
- [100] Martin Vetterli and Didier LeGall. Perfect reconstruction FIR filter banks: some properties and factorizations. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1057–1071, July 1989.
- [101] Joachim E. Vollrath, Tobias Schafhitzel, and Thomas Ertl. Employing complex GPU data structures for the interactive visualization of adaptive mesh refinement data. In *Proceedings Eurographics/IEEE Workshop on Volume Graphics 2006*, pages 55–58,136, 2006.
- [102] Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. Member-Philipp Slusallek.
- [103] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 15–24, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.
- [104] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 7–13, 2000.
- [105] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmerman, and Thomas Ertl. Level-of-detail volume rendering via 3d textures. In *Proceedings IEEE Volume Visualization and Graphics Symposium 2000*, pages 7–13. ACM Press, 2000.
- [106] Rüdiger Westermann. A multiresolution framework for volume rendering. In *1994 Symposium on Volume Visualization*, October 1994.
- [107] G. Wetekam, D. Staneker, U. Kanus, and M. Wand. A hardware architecture for multi-resolution volume rendering. In *Proceedings ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 45–51, New York, NY, USA, 2005. ACM Press.

- [108] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11:201–227, 1992.
- [109] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274. ACM Press, 1978.
- [110] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-weighted color interpolation for volume sampling. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 135–142, 1998.
- [111] Chris Wyman, Steven Parker, Charles Hansen, and Peter Shirley. Interactive display of isosurfaces with global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):186–196, 2006.
- [112] Boon-Lock Yeo and Bede Liu. Volume rendering of DCT-based compressed 3d scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1:29–43, March 1995.
- [113] C. Zhang, D. Xue, and R. Crawfis. Light propagation for mixed polygonal and volumetric data. In *CGI '05: Proceedings of the Computer Graphics International 2005*, pages 249–256, Washington, DC, USA, 2005. IEEE Computer Society.
- [114] Caixia Zhang and Roger Crawfis. Shadows and soft shadows with participating media using splatting. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):139–149, 2003.