Fast High Accuracy Volume Rendering

by

Kenneth Dean Moreland

B.S., Computer Science, New Mexico Institute of Mining and Technology, 1997
B.S., Electrical Engineering, New Mexico Institute of Mining and Technology, 1997
M.S., Computer Science, University of New Mexico, 2000

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July 2004

Dedication

To JoAnn, Ryan, and Mackenzie, without whom I would be bitter and alone.

Acknowledgments

How can I start these acknowledgments without first thanking my wife, JoAnn? For the love, for the care, for the support, and, most importantly, for never letting me forget what is really important, this is for you.

Although I left the nest long ago, much credit to my academic success belongs to my parents. Had you not instilled such a high importance to education in me, I doubt I would have made it here today.

A big thanks goes to my adviser, Edward Angel. Thank you for your endless supply of ideas and advice. Thank you for wading through and correcting this dissertation. Thank you for not retiring yet. I also express gratitude to the rest of my committee members: Thomas Caudell, Philip Heermann, Arthur Maccabe, and Bernard Moret. Thank you for your time, for your support, and for passing me.

I also appreciate all the help I received from my coworkers at Sandia National Laboratories. Philip Heermann, thank you for your managerial support, without which I could never have started on this journey. Brian Wylie and Andrew Wilson, thank you for your technical ideas and support. Patricia Crossno, thank you for your financial support and use of your impressive technical library.

Finally, I feel obliged to thank the Boston Park Plaza Hotel. It was while lying awake in one of its rooms wondering if I would need to evacuate the building for a *third* false fire alarm that I imagined the initial ideas for my thesis. Had it not been for that faulty wiring, who knows what I would have been researching?

The DOE Mathematics, Information, and Computer Science Office funded this research. The National Science Foundation also funded this research under grant number CDA-9503064. The work was performed at Sandia National Laboratories and the University of New Mexico. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Fast High Accuracy Volume Rendering

by

Kenneth Dean Moreland

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July 2004

Fast High Accuracy Volume Rendering

by

Kenneth Dean Moreland

B.S., Computer Science, New Mexico Institute
of Mining and Technology, 1997
B.S., Electrical Engineering, New Mexico Institute
of Mining and Technology, 1997
M.S., Computer Science, University of New Mexico, 2000
Ph.D., Computer Science, University of New Mexico, 2004

Abstract

In computer graphics, color calculations for volumes can be significantly more computationally intensive than that for surfaces. Consequently, until now no interactive volume rendering system performs these calculations for even linearly interpolated luminance and opacity without resorting to rough approximations or a finite set of precomputed values.

In this dissertation, I describe an unstructured grid volume renderer. The renderer is interactive, yet it can produce artifact free images that traditionally would take minutes to render. I employ a projective technique that takes advantage of the

expanded programmability of the latest 3D graphics hardware. I analyze also an optical model commonly used for scientific volume rendering and derive new approximations that are exceptionally accurate but computationally feasible in real time. I demonstrate a system that can accurately produce a volume rendering of an unstructured grid with any piecewise linear transfer function. Furthermore, my algorithm is capable of rendering over 300 thousand tetrahedra per second yet is not limited by pre-integration techniques.

Contents

List of Figures xiv

List of Tables xvii

1 Introduction 1

1.1 Volume Rendering Overview 3

1.2 Proposed Graphics Hardware Extension 5

1.3 Thesis Contribution 9

2 The Volume Rendering Integral 10

2.1 Derivation of the Volume Rendering Integral 11

2.2 Properties of the Volume Rendering Integral 15

2.2.1 Per Wavelength Calculation 15

2.2.2 Glow Parameter 17

2.2.3 Opacity and Blending 18

2.2.4 Piecewise Integration 20

2.3 Closed Forms of the Volume Rendering Integral 22
2.3.1 Only Attenuation 23
2.3.2 Only Emission 24
2.3.3 Completely Homogeneous 24
2.3.4 Homogeneous Particles with Variable Density 25
2.3.5 Linear Interpolation of Volume Parameters 26

3 Practical Implementations of Volume Rendering 29

3.1 Cell-Ray Intersections 30
3.1.1 Ray Casting 31
3.1.2 Cell Projection 35
3.1.3 Rectilinear Grid Resampling 49
3.2 Color Computations 52
3.2.1 Riemann Sum 53
3.2.2 Average Luminance and Attenuation 55
3.2.3 Linear Interpolation of Luminance and Intensity 56
3.2.4 Gaussian Attenuation 60
3.2.5 Pre-Integration 62
3.3 Summary 64

4 Cell Projection 66

4.1 Quick Review of View Independent Cell Projection 66

4.2 GPU-CPU Balanced Cell Projection 70

4.3 Adaptive Transfer Function Sampling 73

4.4 Synopsis 82

5 Ray Integration **84**

5.1 Linear Interpolation of Luminance 85

5.2 Linear Interpolation of Attenuation 86

5.2.1 Computing ζ 87

5.2.2 Computing Ψ 87

5.2.3 Domain of Ψ 89

5.2.4 Resolution of Ψ Table 92

5.3 Linear Interpolation of Opacity 93

5.3.1 Initial Approximation 95

5.3.2 Improved Approximation 97

5.4 Synopsis 100

6 Results and Comparisons **101**

6.1 Speed 101

6.1.1 Cell Projection 102

6.1.2 Ray Integration 107

6.2 Accuracy 109

6.3 Cell Boundary Smoothness 114

7 Conclusions **120**

A Computing Pre-Integration Tables with *Mathematica* **123**

B Vertex and Fragment Programs **125**

B.1 Clipped Tetrahedron Projection Vertex Program 125

B.2 Fragment Program for Clipped Tetrahedron Projection 127

B.3 Volume Rendering Integral with Linear Attenuation and Luminance . 129

B.4 Volume Rendering Integral with Partial Pre-Integration 133

B.5 Volume Rendering Integral with Linear Opacity and Luminance, Rough Approximation 134

B.6 Volume Rendering Integral with Linear Opacity and Luminance, Close Approximation 135

References **136**

Index **147**

List of Figures

1.1 A photorealistic scene with translucent volumes (clouds). 1

1.2 Surface rendering versus volume rendering. 2

1.3 Volume representations. 4

1.4 Different renderings of two hexahedron cells. 6

1.5 Light transport through faces versus solids. 7

1.6 The standard graphics pipeline. 7

2.1 An illuminated volume. 11

2.2 Particle model of volumetric rendering. 13

3.1 Hierarchical pruning for ray casting 32

3.2 Cell traversal ray tracing. 33

3.3 Projected tetrahedra classes. 37

3.4 GATOR basis graph. 39

3.5 GATOR permutations. 40

3.6 Ray-cell–face intersection. 41

3.7 Ray-plane intersections at vertices. 44

3.8 A visibility cycle. 44

3.9 MPVO cell connectivity graph. 46

3.10 Shear-warp factorization. 50

3.11 Rectilinear volume slices. 51

3.12 Error in luminance averaging. 56

3.13 The calculations performed by a pre-integration table lookup. 63

4.1 Effect of aliasing of a transfer function. 74

4.2 Tetrahedron clipping. 81

5.1 Plot of Ψ against $\tau_{\mathrm{b}}D$ and $\tau_{\mathrm{f}}D$. 89

5.2 Plot of Ψ against $\tau_{\mathrm{b}}D$ and $\tau_{\mathrm{f}}D$. 90

5.3 Plot of Ψ against γ_{b} and γ_{f}. 91

5.4 Maximum error of Ψ calculation using a lookup table. 92

5.5 Comparison of ray-integration approaches, linear attenuation. 94

5.6 Approximation of ζ for linearly interpolated opacity. 95

5.7 Approximation of Ψ for linearly interpolated opacity. 96

5.8 Plot of ζ with linearly interpolated α. 97

5.9 Plots the error of the Ψ approximations. 98

5.10 Comparison of ray-integration approaches, linear opacity. 99

6.1 Sample data sets. 103

6.2 Running times of cell projection. 105

6.3 Effect of fragment processing on cell projection speed. 106

6.4 Running times of ray integration. 108

6.5 Error for linear attenuation approximations, $D = 0.001$. 110

6.6 Error for linear attenuation approximations, $D = 0.1$. 111

6.7 Error for linear attenuation approximations, $D = 1$. 111

6.8 Error for linear attenuation approximations, $D = 100$. 112

6.9 Error for linear opacity approximations, $D = 0.001$. 113

6.10 Error for linear opacity approximations, $D = 0.1$. 113

6.11 Error for linear opacity approximations, $D = 1$. 114

6.12 Model used to study Mach bands caused by approximation errors. . 115

6.13 Ganglion receptor response. 116

6.14 Cell boundaries with constant attenuation. 116

6.15 Cell boundaries with large attenuation in the back. 117

6.16 Cell boundaries with large attenuation in the front. 117

6.17 Cell boundaries with constant opacity. 118

6.18 Cell boundaries with large opacity in the back. 118

6.19 Cell boundaries with large opacity in the front. 119

List of Tables

6.1 Data sets used for testing. 102

6.2 Growth in data sets for adaptive transfer function sampling. 104

6.3 Running times of cell projection. 104

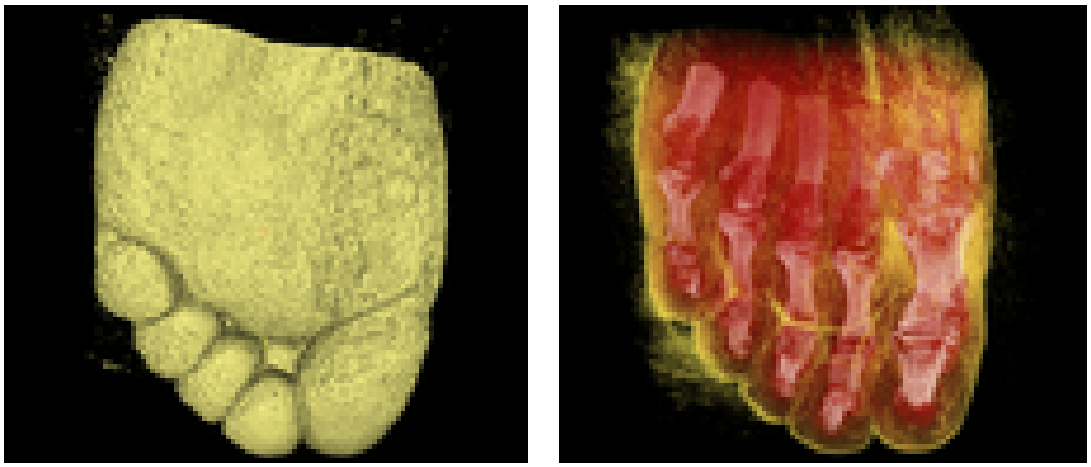
6.4 Running times of ray integration. 107

Chapter 1

Introduction

Even though we call them three-dimensional graphics cards, commodity graphics hardware directly supports only zero-, one-, and two-dimensional primitives (points, lines, and polygons). The reason is simple. An opaque solid object is visually indistinguishable from just its surface when viewed from the outside. However, a photorealistic scene may involve any number of translucent volumetric objects such

Figure 1.1: A photorealistic scene with translucent volumes (clouds). The image is courtesy of Mark Harris [37].



(a) Surface Rendering (b) Volume Rendering

Figure 1.2: Visualization of a rotational x-ray scan of a human foot. The dataset is courtesy of Philips Research, Hamburg Germany.

as clouds, dust, steam, fog, or jiggly food products [38]. We describe the synthesis of such elements as **volume rendering**. Figure 1.1 shows an example usage of translucent objects in a photorealistic scene.

In addition, **direct volume visualization** has become a popular technique for visualizing volumetric data from sources such as scientific simulations, analytic functions, and medical scanners such as MRI, CT, and ultrasound. All these data comprise samples, voxels, or cells distributed in a three dimensional volume. Visualizing these types of data can be problematic. A human being is capable of perceiving only the two dimensional projection of an object on the retina in the back of his eye, and the majority of objects a person sees in day to day life is opaque. Opaque surfaces, therefore, drive many visualization techniques, but the consequence is that interesting features of volumetric data could be lost if embedded in the middle, hidden by outer surfaces such as those demonstrated in Figure 1.2.

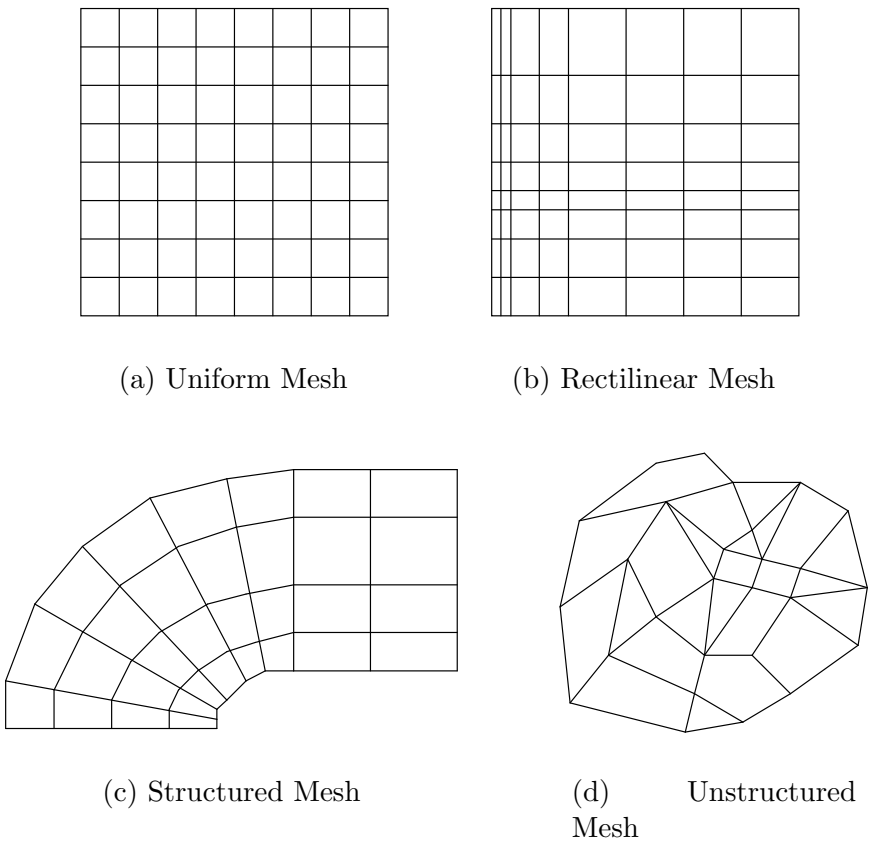
Unfortunately, equations describing all but the simplest and approximate optical

models are difficult to solve in real time. Many volume-rendering applications use drastic simplifications such as constant light emission or absorption (or both) through discrete segments. A scarce few use linear interpolation of both. Meanwhile, many organizations, including Sandia National Laboratories, have a continuing interest in **unstructured meshes**, volumetric models that can, and often do, vary wildly in cell size, shape, and connectivity. Although most of these models have **linear cells**, cells that vary linearly in both position and parameter, there is also a growing interest in models with **nonlinear cells**, cells with nonlinear parametric functions defining their shape and parameters [42]. Currently no interactive direct volume visualization systems can render such elements correctly.

This dissertation seeks to improve the current state of the art of volume rendering. In it, I demonstrate how to render a model consisting of linear cells (or, equivalently, a first order (linear) approximation). My method for volume rendering will be fast enough for interactive applications, whereas other systems may take minutes for a single rendered image. My method performs calculations close to those defined by the volumetric model I use, whereas others make brash approximations. Furthermore, unlike other systems, my method does not require any preprocessing, which will allow for fast changes in volume rendering parameters such as the transfer function.

1.1 Volume Rendering Overview

In this section, we review the fundamental concepts of volume rendering. First, we discuss how we can represent a volume. In scientific visualization, as in many other fields of visualization, volumes are most often represented as a **mesh** (sometimes known also as a **grid**). A mesh is a collection of volumetric elements called **cells**. The cells themselves are defined over a set of points called **vertices**. **Geometry** and **topology** define a mesh. The geometry describes the layout of vertices in space.



(a) Uniform Mesh

(b) Rectilinear Mesh

(c) Structured Mesh

(d) Unstructured Mesh

Figure 1.3: Various mesh types used to represent volumes. For clarity, I show 2D meshes here, but the differences among mesh types hold for any number of dimensions.

The topology (sometimes referred to also as the **connectivity**) connects vertices to form cells.

Figure 1.3 shows four different classifications of meshes. A **uniform mesh** has both the topology and geometry fixed such that the vertices are in an orthographic grid and the cells are the axes aligned boxes formed by the vertices. In three dimensions, these cells are **voxels**. A **rectilinear mesh** is the same as a uniform mesh except that geometry is relaxed such that the spacing of the vertices may change. A general **structured mesh** has the same topology of a regular grid, but the geom-

etry is free to place the vertices anywhere in space so long as the topology remains consistent. An **unstructured mesh** is nonrestrictive. It is free to have any type of geometry or topology.

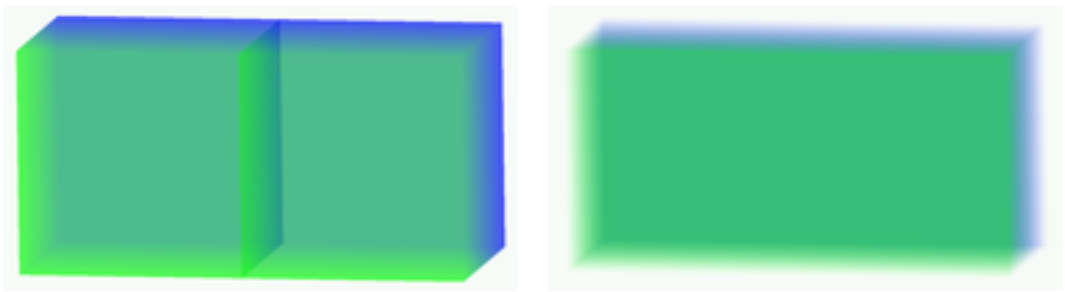
To synthesize an image of a volume, a rendering system first has to determine which cells in the volume contribute to which parts of the image. There are two general approaches. The first approach is **ray casting**, where the rendering system casts rays from the viewpoint through each pixel of the image into the volume. The second approach is **cell projection**, where the rendering system projects each cell onto the viewing plane.

Once it has determined the location of each part of the volume in the viewing plane, the rendering system must compute the intensity of light emitted by the volume. I discuss this computation in detail in Chapter 2.

1.2 Proposed Graphics Hardware Extension

Commodity graphics hardware supports blending operations that enable translucent polygons. Thus, a naïve approach to rendering translucent volumes might be simply to render the faces that make up the model of the volume as translucent polygons. However, the difference between rendering a translucent solid object versus the translucent surfaces of an object is subtle but important. Getting it wrong can lead to significant artifacts as shown in Figure 1.4.

Figure 1.5 demonstrates in two dimensions why face rendering is different from solid rendering. We can easily extrude the triangles to 3D prisms for an equally valid demonstration in three dimensions. When rendering only faces, as shown in Figure 1.5(a), each ray of light passing through the triangle crosses exactly two faces of the triangle. The result is that the contribution of the triangle is consistent



(a) Hollow cells with translucent faces. (b) Solid, translucent cells.

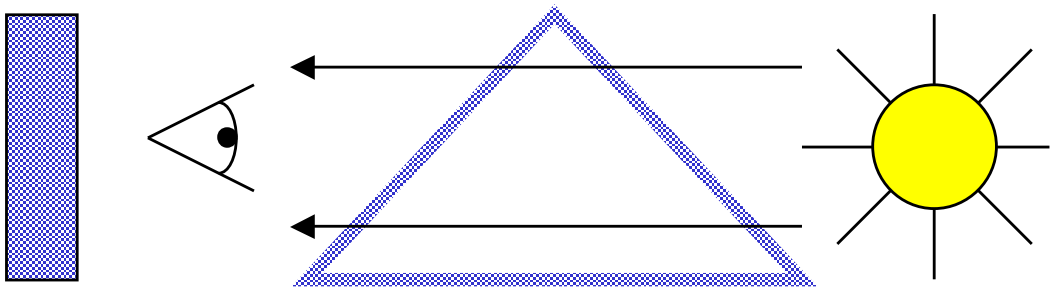
Figure 1.4: Different renderings of two hexahedron cells. The image on the left is generated by rendering the faces as translucent polygons. The image on the right is a true rendering of a solid volume.

throughout its projection on the viewing plane.

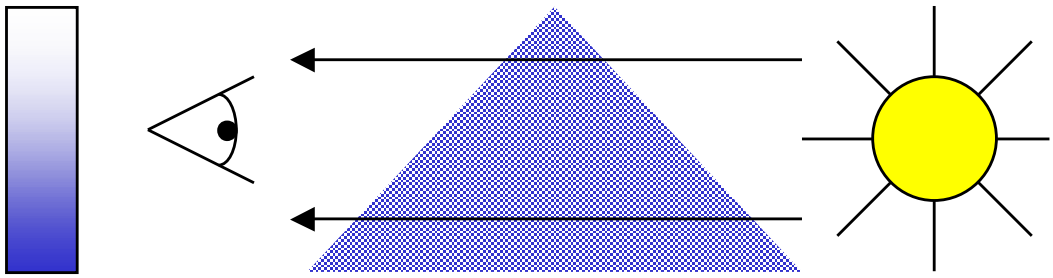
In contrast, when rendering solids, as shown in Figure 1.5(b), light must pass through more of the solid at its thickest parts. Therefore, the contribution of the triangle is much greater where it is thick and negligible where the corners taper off. The result is a nonlinear change in color across the viewing plane. I shall discuss the light transport through volumes in detail in Chapter 2.

Given the interest in volumetric rendering, can we modify the popular graphics pipeline to render such objects? Surprisingly, yes. For roughly a decade, applications have used texture hardware to render rectilinear volumes [6, 12, 106] while others have used polygon rendering hardware to speed up the rendering of unstructured tetrahedra [86]. More recently, King, Wittenbrink, and Wolters [45] proposed (but never implemented) a graphics pipeline architecture capable of rendering translucent tetrahedra. A year later, both Wylie [110] and Weiler [98, 100], each with their respective colleagues, used commodity graphics hardware with programmable shaders to render tetrahedra directly in the pipeline.

Figure 1.6 shows a simplified diagram of the standard **graphics pipeline**. Al-



(a) Light transport through a hollow triangle with translucent edges.



(b) Light transport through a solid, translucent triangle.

Figure 1.5: Demonstration of light transport through faces versus light transport through solids. The material properties are homogeneous throughout the faces or solid. The attenuation and emission of light is constant through the faces but varies based on the thickness of the solid.

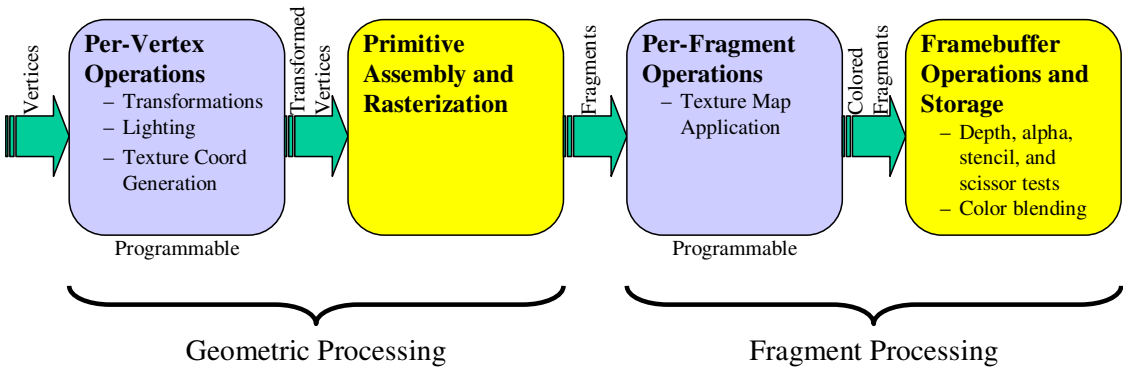


Figure 1.6: The standard graphics pipeline.

most all commodity 3D graphics hardware today implements this pipeline.[1] At the left, **vertices** of **primitives** (points, lines, and polygons) enter the pipeline. Each vertex has associated with it several properties such as position, material, and texture coordinates. The first unit of the pipeline transforms, lights, and clips each vertex independently. The second unit assembles the transformed vertices into primitives and **rasterizes** them. That is, it projects the primitives onto the viewing plane and samples them. We call the samples generated **fragments**. Fragments are like vertices in that they carry along information such as color and texture coordinates. The next unit independently applies the final color to each fragment. The final unit blends the results into the appropriate **pixel** of the **frame buffer**, a rectangular array storing color, depth, and other information of an image.

As can be seen, we can split operations into two major categories: **geometric processing**, which determines which primitives are intersected by each viewing ray, and **fragment processing**, which assembles material properties of intersected primitives along a ray to determine the final color of a pixel. Until recently, this pipeline was fixed. Fortunately, the latest releases of graphics cards have flexible programmable units for both geometric and fragment processing [59, 61, 66, 77].

The tasks of volume rendering can also be divided into geometric and fragment processing. Previous work [45, 98, 110] has shown that the geometric processing for 2D primitives needs to be tweaked little to support 3D primitives. The fragment processing of 2D and 3D primitives also can be similar, but the computation for 3D primitives is significantly greater. The color computation of a ray segment through a 3D cell must take into account color and opacity changes throughout the segment and integrate all these values. Consequently, most implementations perform a rough approximation rather than perform the actual integration.

[1] The only exceptions are graphics boards for specialized rendering such as volume rendering rectilinear grids [14, 53, 67, 73, 78].

1.3 Thesis Contribution

This dissertation contributes to the speed and accuracy with which we can perform the geometric and fragment processing for volume rendering. I demonstrate a technique that is fast enough to perform in real time yet accurate. Furthermore, we can use the technique directly on graphics hardware.

I organize the rest of this dissertation as follows. In Chapter 2, I present the model used to describe light transport through a volume and derive equations that compute colors of viewing rays. In Chapter 3, I briefly describe other volume rendering systems and describe how they solve the various problems associated with volume rendering. I then follow by introducing improvements to hardware based cell projection in Chapter 4 and improvements to volume ray integration in Chapter 5. Finally, I present results of the implementation in Chapter 6 and draw conclusions in Chapter 7.

Chapter 2

The Volume Rendering Integral

Before rendering a translucent volume, we need to understand how such a volume transports light. To this end, we will build an **optical model**. The optical model describes the light transport within the volume and allows us to define the behavior of light passing through the volume.

Many researchers began building optical models in the early 1980's to synthesize photorealistic images with volumetric elements such as clouds [4, 40, 62]. By 1988, others were building models to perform volume rendering for scientific visualization [20, 83]. Williams and Max [104] later refined the approach for use with various interpolation functions and cell shapes.

In this chapter, I discuss the **volume rendering integral**. The volume rendering integral is an equation that computes the color of light that passes through a volume. I first derive the volume rendering integral using a model and derivation similar to that of Max [63]. I then discuss properties of this equation that are important for practical applications. Finally, I present several closed forms that other researchers have developed.

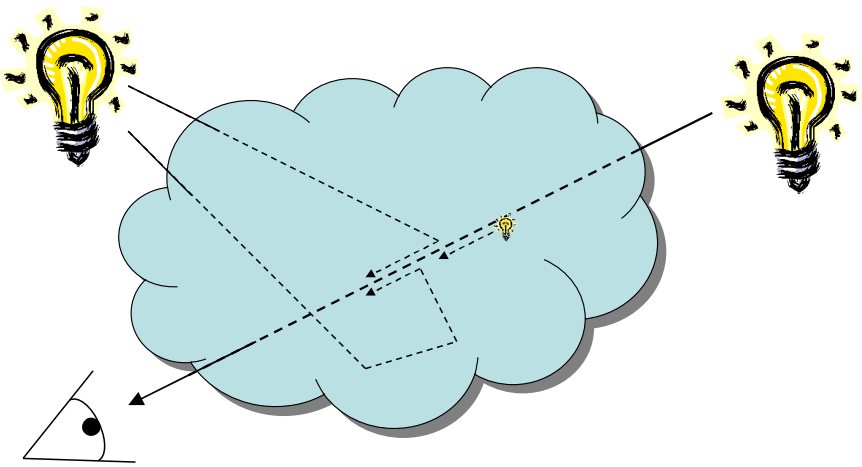


Figure 2.1: An example of an illuminated volume.

2.1 Derivation of the Volume Rendering Integral

Consider a translucent volumetric cloud such as the one depicted in Figure 2.1. What contributes to the color of the cloud? Some light originating from behind the cloud may pass right through it, although the cloud will most likely attenuate the light. Light may originate from within the cloud itself if the cloud contains glowing material. The cloud may diffusely reflect light of other sources from within. The diffuse reflection can disperse the light through the volume causing a **scattering** effect.

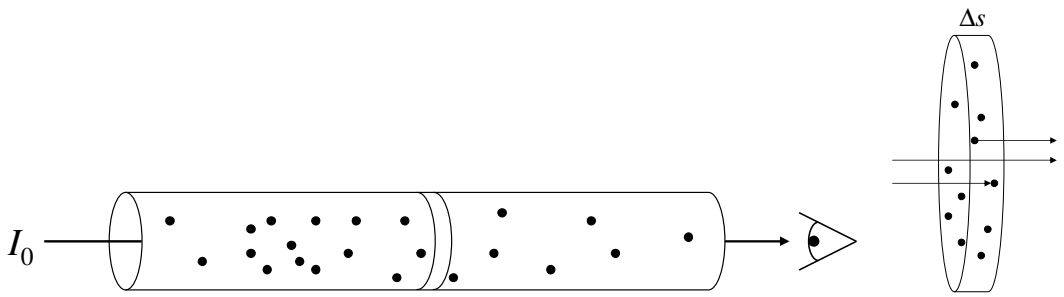
We start by modeling the volume as a space filled with minute particles. Each particle is opaque and occludes light waves, but is itself too small to see individually. In addition to occluding light, each particle emits light also. The emitted light may be an energy generated by the particle itself, such as from a glowing ember, or light diffusely reflected from a different particle or another light source. We treat either case the same, thereby abstracting away the diffuse emission calculation.

Let us pause a moment to consider the ramifications of the previous statement.

By neglecting to consider the effects of diffuse lighting, we potentially miss important lighting effects. We are not taking into account **shadowing**, the attenuation of light between a particle and a light source. We are neglecting **multiple scattering**, the illumination of a particle by light rays reflected off other particles, also. We use the approximation to make the calculation more tractable by considering only light that passes directly between a particle and the viewpoint. Because the approximation allows volume rendering to occur at interactive rates and the shadowing and scattering effects do not necessarily make scientific visualization clearer [71], this approximation has been used extensively in volume visualization since Sabella introduced it in 1988 [83].

Furthermore, although the model we describe does not calculate shadowing or scattering directly, it still allows us to do such calculations. Recall that the model allows each particle to give off any amount of light energy. If we compute the total light energy generated by the particles and reflected off the particles in a secondary calculation, we can plug the result into the particle emission parameter of this model. This approach of using two or more simplified models to generate multiple lighting effects has already proved to be an effective approach to performing shadowing and scattering with opaque surfaces [10, 18, 41, 85]. In fact, many recent approaches to global volume illumination [19, 37, 48, 51, 112, 113] perform a two-step approach of first computing particle colors and then integrating ray segments.

By considering only local lighting effects, we can simplify our analysis of volumetric lighting computations by taking into account only a single long cylinder centered on the viewing ray that passes through the volume as shown in Figure 2.2(a). The cylinder is thin enough to assume that volume properties do not change across its breadth, but they may (and probably will) change across its length. At the back end of the cylinder, background light comes in, and at the front end of the cylinder, light exits and travels to the user's eye. The light intensity coming from the front end of



(a) A cylindrical cutout of the volume

(b) A disk of length Δs.

Figure 2.2: Particle model of volumetric rendering.

the cylinder will determine the color value of one pixel.

Let the cross sectional area of the cylinder be E. Now consider a thin slab of this cylinder whose base is also of area E and whose length is Δs. As shown in Figure 2.2(b), as light rays pass through this disk, particles obstruct some rays whereas other rays pass straight through. Still other light rays originate from particles in the disk. Let A be the cross sectional area of each particle, ρ be the density of particles per unit volume, and L be the light emission of the particles per projected area in the direction of the ray. Both ρ and L may vary over the volume. Each disk has volume $E\Delta s$, and therefore contains $N = \rho E \Delta s$ particles.

As Δs approaches zero, the overlap of particles becomes zero. At this point, the area of the cylinder obscured by particles is $AN = A\rho E\Delta s$. The fraction of light occluded when flowing through the disk (*i.e.* the fraction of cross sectional area with particles in it) is $AN/E = A\rho\Delta s$. Let us define the **attenuation coefficient** $\tau = A\rho$, which expresses the expected amount of incoming light that is extinguished per unit length (given negligible particle overlap).

In addition to absorbing light, the particles emit light with intensity L per pro-

jected area. L expresses the **luminance** (per wavelength) of the volume. The light emission of each particle is, therefore, LA, and the overall light emission within the disk is $LAN = LA\rho E\Delta s$. The light emitted from the disk goes through the base of area E, so the light emitted per unit area is $LA\rho\Delta s = L\tau\Delta s$.

Given these parameters of the volume density, we can express the behavior of the **intensity** of a ray of light through the volume as

$$\frac{dI}{ds} = L(s)\tau(s) - I(s)\tau(s) \tag{2.1}$$

That is, the change in intensity of the light ray, $I(s)$, as it passes through the volume is equal to the light emitted from the cross section at s minus the amount the incoming light is attenuated. We can solve this differential equation as follows. First, we bring the $I(s)\tau(s)$ term over to the right hand side.

$$\frac{dI}{ds} + I(s)\tau(s) = L(s)\tau(s)$$

Then we multiply both sides by $\exp\left(\int_0^s \tau(t)dt\right)$.

$$\begin{aligned}
\frac{dI}{ds}e^{\int_0^s \tau(t)dt} + I(s)\tau(s)e^{\int_0^s \tau(t)dt} &= L(s)\tau(s)e^{\int_0^s \tau(t)dt} \\
\frac{d}{ds}\left(I(s)\right)e^{\int_0^s \tau(t)dt} + I(s)\frac{d}{ds}\left(e^{\int_0^s \tau(t)dt}\right) &= L(s)\tau(s)e^{\int_0^s \tau(t)dt} \\
\frac{d}{ds}\left(I(s)e^{\int_0^s \tau(t)dt}\right) &= L(s)\tau(s)e^{\int_0^s \tau(t)dt}
\end{aligned}$$

Finally, we integrate everything from $s = 0$ at the back end of the volume to $s = D$ at the eye,

$$I(D)e^{\int_0^D \tau(t)dt} - I_0 = \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

and do one last rearrangement of terms.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \tag{2.2}$$

We call this equation the **volume rendering integral**. An examination verifies that Equation 2.2 appeals to our intuitive sense of what happens to light as it passes through a translucent volume. The first term calculates the amount of incoming light, I_0, that reaches the end of the volume. We see that the incoming light attenuates exponentially with length D, just as we observe from objects at various distances on a foggy day. The second term adds the amount of light emitted at each point along the ray, taking into account the amount of attenuation from each point to the end of the ray.

2.2 Properties of the Volume Rendering Integral

Before continuing, I shall address several details of the volume rendering integral.

2.2.1 Per Wavelength Calculation

I give the volume rendering integral (Equation 2.2 derived in Section 2.1 on page 14) as a scalar function. That is, I define the input luminance and attenuation variables as single scalar values, as is the output light intensity. Although such a calculation is correct for a monochromatic image, most images will benefit from full color.

Visible light actually comprises a continuous spectrum of light waves with varying wavelengths [1]. Each wavelength stimulates the receptors in the human eye differently. A cloud attains its color by having different responses to each wavelength of light. It is therefore often practical to define the luminance and attenuation properties of a volume on a per wavelength basis.

Given volume properties defined on a per wavelength basis, how does this affect the volume rendering integral? As light travels through or bounces off a material,

it is generally true that the wavelength of a given ray of light does not change and that one ray of light has no effect on other rays of light. Thus, the produced light intensity of a particular wavelength, I_λ, depends only on volumetric properties for that particular wavelength, L_λ and τ_λ. Therefore, technically we should express the volume rendering integral as

$$I_\lambda(D) = I_{\lambda 0} e^{-\int_0^D \tau_\lambda(t)dt} + \int_0^D L_\lambda(s)\tau_\lambda(s)e^{-\int_s^D \tau_\lambda(t)dt}ds \tag{2.3}$$

However, because we have defined everything on a per wavelength basis, and because all calculations are wavelength independent, the λ subscript is redundant, and I drop it from further equations.

Given this "new" per wavelength form of the volume rendering integral, a question arises: For how many wavelengths should we compute the volume rendering integral? It is impractical and unnecessary to perform the calculation for the continuous spectra of visible light. Furthermore, because speed is paramount, the fewer wavelengths we have to compute the better.

The answer is three. Any color space needs only three parameters to represent any color perceptible by humans. The color photoreceptors of the human eye come in only three flavors, and we perceive color by the amount each receptor type is stimulated [26]. Therefore, we can combine light comprised of only three different wavelengths with different intensities to reproduce any color discernible with the human eye. The wavelengths for red, green, and blue light are a good choice [27]. This is the **RGB color space**.

In addition to representing colors in RGB color space, I shall perform calculations only on red, green, and blue wavelengths. Although we can represent every color with only red, green, and blue wavelengths, a volume may have material properties that we do not capture on only these wavelengths. This can introduce inaccuracies. For example, although equal parts of red and green light are perceived as yellow, a ray of

yellow light passing through a volume may behave differently than equal parts of red and green. Although there has been a bit of exploration in capturing these effects [3, 35], the effects are of limited value and most research (including this dissertation) ignores them.

2.2.2 Glow Parameter

When deriving the volume rendering integral in Section 2.1, I parameterized the emission of light within the volume as the luminance, $L(s)$, which has units per area of visible surface. The actual intensity of light, therefore, varies with respect to the density of particles. Other literature (such as [104]) instead defines the emission of light within the volume with a **glow** parameter, $g(s)$, that varies the light intensity independent of the particle density. We can express the relationship between luminance and glow as

$$g(s) = L(s)\tau(s) \tag{2.4}$$

Modifying the volume rendering integral (Equation 2.2) to use the glow term is a simple matter of substitution.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D g(s) e^{-\int_s^D \tau(t)dt} ds \tag{2.5}$$

Equations 2.2 and 2.5 are equivalent and the choice between them is mostly a matter of preference. In this dissertation, I usually choose Equation 2.2 with the luminance term. When defining volume lighting parameters it is far more natural for the light emission to fluctuate with the particle density.

Changing the particle density independent of the glow can lead to unexpected visual results. Raising the particle density without raising the glow leads to a dark, sooty-looking volume. Lowering the particle density without lowering the glow results

in an overly bright volume, often saturating the color channels of the display device. However, when we parameterize the color by luminance rather than glow, the color of the volume will appear constant as the particle density is varied.

The only real advantage of using the glow parameter is to allow volumes to emit light without attenuating a significant amount of light. This can happen in a hot, tenuous gas such as in a neon sign. This situation is difficult to model with luminance because as the attenuation goes to zero the luminance must go to infinity, but the glow can remain at a finite value. However, this is a rather simple special case to solve (and I do it in Section 2.3.2).

2.2.3 Opacity and Blending

Direct volume rendering by its very nature deals with transparent objects. As such, it is important to understand how to mix a transparent volume with other objects within a scene. We refer to the process of mixing two overlaid images together as **image blending** or **image compositing**.

Porter and Duff [74] in a seminal paper introduce an algebra for image blending; this algebra is still the foundation of compositing in computer graphics today. Therefore, it is important to understand how the volume rendering integral relates to the Porter and Duff algebra to perform blending appropriately.

In brief, Porter and Duff introduce the α blending term (the A part of a standard OpenGL RGBA pixel color), which gives the fraction of a pixel that is covered and will occlude whatever is "behind" the pixel. Another name for the fraction of a pixel covered is the **opacity**. The opacity fits well with the model defined in Section 2.1 with minute particles that occlude light from behind them. Although I show the derivation for the output light intensity from a volume, I do not give the opacity for the volume there. I derive the opacity for the volume here.

Consider the same thin cylinder surrounding the viewing ray shown in Figure 2.2 on page 13. As is shown in the derivation starting on page 13, the fraction of a cross sectional disk of small length Δs occluded by particles is $\tau \Delta s$. Therefore, we can express the transient change in opacity of the cylinder as

$$\frac{d\alpha}{ds} = \tau(s) - \alpha(s)\tau(s) \tag{2.6}$$

That is, the opacity increases by the fraction occluded in the new disk (the first term) that is not already occluded by the cylinder behind it (the second term).

We can solve this differential equation with the same approach used in the derivation of Section 2.1.

$$\begin{aligned}
\frac{d\alpha}{ds} &= \tau(s) - \alpha(s)\tau(s) \\
\frac{d\alpha}{ds} + \alpha(s)\tau(s) &= \tau(s) \\
\frac{d\alpha}{ds} e^{\int_0^s \tau(t)dt} + \alpha(s)\tau(s)e^{\int_0^s \tau(t)dt} &= \tau(s)e^{\int_0^s \tau(t)dt} \\
\frac{d}{ds}\left(\alpha(s)\right) e^{\int_0^s \tau(t)dt} + \alpha(s)\frac{d}{ds}\left(e^{\int_0^s \tau(t)dt}\right) &= \tau(s)e^{\int_0^s \tau(t)dt} \\
\frac{d}{ds}\left(\alpha(s)e^{\int_0^s \tau(t)dt}\right) &= \tau(s)e^{\int_0^s \tau(t)dt}
\end{aligned}$$

At this point, we can integrate both sides of the equation from $s = 0$ to $s = D$.

$$\alpha(D)e^{\int_0^D \tau(t)dt} - \alpha_0 = \int_0^D \tau(s)e^{\int_0^s \tau(t)dt}ds$$

Because we are interested solely in the opacity of the ray segment within the volume and not that behind it, we can assume that $\alpha_0 = 0$ and drop it from the equation. Solving the rest of the equation, we find that

$$\alpha(D) = 1 - e^{\int_0^D \tau(t)dt} \tag{2.7}$$

Although this derivation relies on using α as the function that is the opacity for a given length through the volume, in practice we are interested only in the opacity of

the entire segment. It is therefore often convenient to drop the functional notation and simply refer to the opacity of a given segment as α.

Now that we have formally defined the α term, we can plug it into the volume rendering integral (Equation 2.2 on page 14).

$$I(D) = I_0(1-\alpha) + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \tag{2.8}$$

If we let $\int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds$ be color A, α be opacity A, and I_0 be color B, then we find that Equation 2.8 is really the Porter and Duff A over B operation or, equivocally, the B under A operation, which means we can easily use graphics hardware to do this blending.

2.2.4 Piecewise Integration

As we shall see shortly in Section 2.3, solving the volume rendering integral for all but the simplest functions for luminance and attenuation is difficult to impossible. In practice, we just perform piecewise integration. Most commonly, we segment viewing rays by the model cells that they intersect, and we integrate piecewise per segment. These segments may or may not be of uniform length, usually depending on the model rendered.

Because of the importance of piecewise integration, I will speak to how we may perform piecewise integration. Consider the volume rendering integral given in Equation 2.2, repeated here for convenience.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

Let us examine what happens when we break the integrals into two segments, one in the range $[0, x]$ and the other in the range $[x, D]$. Without loss of generality, we

can rewrite Equation 2.2 as

$$I(D) = I_0 e^{-\int_0^x \tau(t)dt - \int_x^D \tau(t)dt} + \int_0^x L(s)\tau(s)e^{-\int_s^x \tau(t)dt - \int_x^D \tau(t)dt}ds + \int_x^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \quad (2.9)$$

We can rearrange Equation 2.9 by factoring out the constant $e^{-\int_x^D}$ terms.

$$I(D) = \left(I_0 e^{-\int_0^x \tau(t)dt} + \int_0^x L(s)\tau(s)e^{-\int_s^x \tau(t)dt}\right)e^{-\int_x^D \tau(t)dt} + \int_x^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \quad (2.10)$$

Notice that the part of Equation 2.10 in the parenthesis is equal to $I(x)$.

$$I(D) = I(x)e^{-\int_x^D \tau(t)dt} + \int_x^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \quad (2.11)$$

Finally, we can do a change of variables that will place the integral in the range $[0, D']$, where $D' = D - x$.

$$I'(D') = I(x)e^{-\int_0^{D'} \tau'(t)dt} + \int_0^{D'} L'(s)\tau'(s)e^{-\int_s^{D'} \tau'(t)dt}ds \quad (2.12)$$

Careful inspection reveals that Equation 2.12 is equivalent to the original volume rendering integral (Equation 2.2) for the segment closer to the eye when using the intensity of light emanating from the farther segment as the incoming light. Using this relationship, we can perform **back to front rendering**.

In Section 2.2.3, I show that you could express the opacity as $\alpha = 1 - e^{\int_0^D \tau(t)dt}$ and how you can use the Porter and Duff [74] over and under operations described in Section 2.2.3 to blend. The same blending is valid for this piecewise integration, and we can therefore perform it easily on graphics hardware.

We can rearrange Equation 2.9 yet again.

$$I(D) = I_0 e^{-\int_x^D \tau(t)dt} e^{-\int_0^x \tau(t)dt} + e^{-\int_0^x \tau(t)dt} \int_0^x L(s)\tau(s)e^{-\int_s^x \tau(t)dt} ds + \int_x^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt} ds \quad (2.13)$$

Equation 2.13 shows us how to perform **front to back rendering**. Using only information from the front segment (the segment closest to the viewer), you could solve the third term of Equation 2.13. Using this and the opacity of the front segment, you could use the Porter and Duff [74] over operator, described in Section 2.2.3 to blend with the back segment, solving the second two terms of Equation 2.13, and combine the two opacities to allow for another over operation with the incoming I_0 light when it becomes available.

2.3 Closed Forms of the Volume Rendering Integral

In Section 2.1, I provide a model for light transmission through volumes and derive the volume rendering integral (Equation 2.2), provided here for reference.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt} ds$$

However, this equation has no closed form, and we cannot solve it without further information about $L(s)$ and $\tau(s)$. In this section, I will impose various restrictions on $L(s)$ and $\tau(s)$ that will allow us to solve the volume rendering integral.

2.3.1 Only Attenuation

Our first restriction on the volumetric cloud is that it only attenuates light. The cloud emits no light of its own. The only light, if any, reaching the front of the volume comes from light that enters from the back. Mathematically, $L(s) = 0$. Under this restriction, the volume rendering integral reduces to

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} \tag{2.14}$$

The major advantage of this form is its simplicity. The integral $\int_0^1 \tau(t)dt$ can be solved for almost any desired interpolation of attenuation. In addition, when rendering a rectilinear grid with this optical model, we can use the Fourier projection-slice theorem and the fast Fourier transform to render a grid of size $O(N^3)$ in $O(N^2 \log N)$ time [60].

Furthermore, consider what happens when we segment the volume rendering integral to perform piecewise integration.

$$\begin{aligned} I(D) &= I_0 e^{-\int_0^{x_1} \tau(t)dt - \int_{x_1}^{x_2} \tau(t)dt - \cdots - \int_{x_{n-1}}^{x_n} \tau(t)dt} \\ &= I_0 e^{-\int_0^{x_1} \tau(t)dt} e^{-\int_{x_1}^{x_2} \tau(t)dt} \cdots e^{-\int_{x_{n-1}}^{x_n} \tau(t)dt} \end{aligned} \tag{2.15}$$

Because multiplication is commutative, Equation 2.15 shows that we can compute and then combine in any order the integrals for each segment to obtain a correct image. This allows us to render the cells in an **order independent** fashion. Because cell **visibility sorting** can be a challenging and computationally intensive process [11, 52, 54, 64, 90, 91, 92, 103], this can be a great advantage when using projective methods.

Unfortunately, the only-attenuation model has major weaknesses. Because the model emits no light, the amount of lighting effects is severely limited. The order independence property that makes this model so easy to render means also that there

are no depth cues. There is no way to tell, given a fixed viewpoint, if one feature is in front of another. Totsuka and Levoy [93] offer techniques to introduce visual cues, but the cues are only moderately effective and are not realistic.

2.3.2 Only Emission

Our next restriction is the opposite of the previous one. We now assume that the volume emits light, but the attenuation is negligible. With this assumption, defining the emission in terms of luminance, which is in units of per particle area, is not practical as the particle area is zero. Therefore, rather than use Equation 2.2 as the form for the volume rendering equation, we will instead use Equation 2.5 defined in Section 2.2.2 that uses a glow parameter instead of luminance. Setting $\tau(s) = 0$, Equation 2.5 reduces to

$$I(D) = I_0 + \int_0^D g(s)ds \tag{2.16}$$

The only-emission model has all the same advantages as the absorption-only model (defined in Section 2.3.1), but it shares all the same disadvantages also. In addition, the only-emission model suffers from color saturation. Equation 2.16 has no bounds on the intensity of the final light ray, and in practice, the intensity can easily soar beyond what a display device can handle. Therefore, real volume rendering systems seldom use the only-emission model.

2.3.3 Completely Homogeneous

Another simple approximation is to assume that the volume is homogeneous. That is, the attenuation and luminance parameters do not vary. We can model this by

substituting the constants L and τ for $L(s)$ and $\tau(s)$, respectively, in Equation 2.2.

$$\begin{aligned} I(D) &= I_0 e^{-\int_0^D \tau dt} + \int_0^D L\tau e^{-\int_s^D \tau dt} ds \\ &= I_0 e^{-\tau D} + \int_0^D L\tau e^{-\tau(D-s)} ds \\ &= I_0 e^{-\tau D} + L\, e^{-\tau(D-s)}\Big|_{s=0}^{D} \\ &= I_0 e^{-\tau D} + L\left(1 - e^{-\tau D}\right) \end{aligned} \tag{2.17}$$

It is, of course, not practical or interesting to limit our volume to be completely homogeneous. Instead (as alluded to in Section 2.2.4), you can use Equation 2.17 in a Riemann sum to accurately estimate the volume rendering integral by breaking it up into small enough pieces [63].

If we sample the volume uniformly, D in Equation 2.17 is constant. In this case, we can convert the attenuation parameter (τ) to an opacity (α) offline and use Porter and Duff blending as described in Section 2.2.3 to perform this Riemann sum. Stein, Becker, and Max [92] demonstrate how to use 2D texture hardware to convert the attenuation and distance to opacity before blending for volumes not sampled uniformly.

Although, technically, we could subdivide our volume fine enough for any amount of accuracy (although quantization errors become a problem), more subdivisions result in more computational overhead. Less constrained forms of the volume rendering integral can lead to greater accuracy with fewer subdivisions.

2.3.4 Homogeneous Particles with Variable Density

Recall from Section 2.1 that we modeled our volume as a collection of minute particles. Max, Hanrahan, and Crawfis [64] proposed the following restriction. Let the

density of the particles, ρ, vary throughout the volume, but constrain the particles to all have the same properties.

The luminance, L, of the volume is a direct property of the volume and is constant. In Section 2.1, I define the attenuation coefficient as $\tau = A\rho$ where A is the cross sectional area of the particles. A is a property of the particles, which is constant when the particles are homogeneous. Thus, τ is proportional to ρ.

It is therefore equivalent to say that τ varies whereas L does not. We therefore constrain the volume rendering integral by substituting L for $L(s)$ in Equation 2.2.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L\tau(s) e^{-\int_s^D \tau(t)dt} ds$$

Assuming that $\tau(s)$ is integrable, we can further resolve this equation.

$$\begin{aligned} I(D) &= I_0 e^{-\int_0^D \tau(t)dt} + L e^{-\int_s^D \tau(t)dt}\Big|_{s=0}^{D} \\ &= I_0 e^{-\int_0^D \tau(t)dt} + L\left(1 - e^{-\int_0^D \tau(t)dt}\right) \end{aligned} \tag{2.18}$$

Again, we have a more powerful but less than ideal form to the volume rendering integral. Here we are able to vary the density of our cloud in almost any fashion we desire, yet the luminance must remain constant.

2.3.5 Linear Interpolation of Volume Parameters

So far, in all the closed forms to the volume rendering integral that I have presented, none is capable of interpolating both the luminance and attenuation parameters of the volume (without the use of piecewise integration). The simplest form of interpolation is linear interpolation. Williams and Max [104] were the first to solve the volume rendering integral with linear interpolation of both luminance and attenuation. They choose to parameterize their equations using a tetrahedron that a viewing ray intersects.

I instead give a closed form that parameterizes the volume rendering equation using only $L(s)$ and $\tau(s)$, which, in my humble opinion, leads to a simpler form of the equation. Without loss of generality, I use the following linear forms of $L(s)$ and $\tau(s)$.

$$L(s) = L_{\mathrm{b}}\frac{D-s}{D} + L_{\mathrm{f}}\frac{s}{D} \tag{2.19}$$

$$\tau(s) = \tau_{\mathrm{b}}\frac{D-s}{D} + \tau_{\mathrm{f}}\frac{s}{D} \tag{2.20}$$

In Equation 2.19, L_{f} and L_{b} describe the luminance at the front and back of the ray, respectively. This is likewise for the attenuation in Equation 2.20.

Substituting Equation 2.19 and Equation 2.20 into Equation 2.2 results in a solvable equation, although the calculus to do so is difficult. Using the help of a mathematical solver such as *Mathematica* [108], we get

$$\begin{aligned} I(D) = I_0 e^{-D\frac{\tau_{\mathrm{b}}+\tau_{\mathrm{f}}}{2}} + L_{\mathrm{f}} - L_{\mathrm{b}} e^{-D\frac{\tau_{\mathrm{b}}+\tau_{\mathrm{f}}}{2}} \\ + (L_{\mathrm{b}} - L_{\mathrm{f}})\frac{1}{\sqrt{D(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}} e^{\frac{D}{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}\tau_{\mathrm{f}}^2}\sqrt{\frac{\pi}{2}} \\ \left[\mathrm{erf}\left(\tau_{\mathrm{b}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}}\right) - \mathrm{erf}\left(\tau_{\mathrm{f}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}}\right)\right] \end{aligned} \tag{2.21}$$

Equation 2.21 has many terms, which makes it computationally intensive to compute. Furthermore, there are instances of the erf function. Here, erf is the **error function**, defined as $\mathrm{erf}(x) = \frac{2}{\pi}\int_0^x e^{-u^2}du$. The erf function does not have a closed form, but there are several known numerical methods to compute it with sufficient accuracy.

Further analysis shows us that if $\tau_{\mathrm{b}} < \tau_{\mathrm{f}}$, Equation 2.21 contains imaginary terms, that is, terms with $\mathbf{i} = \sqrt{-1}$ in them. The idea of having complex values for light intensity is a bit disturbing, and it might lead you to question the validity of Equation 2.21. However, as long as the values for τ_{b}, τ_{f}, L_{b}, and L_{f} are real, all imaginary terms will cancel out.

An implementation evaluating Equation 2.21 could compute it by performing complex arithmetic. However, because it is generally more convenient to perform strictly real arithmetic, we can manipulate Equation 2.21 to have only real numbers when $\tau_\mathrm{b} < \tau_\mathrm{f}$. We can do this using the **imaginary error function**, erfi, defined as $\mathrm{erfi}(x) = \mathrm{erf}(\mathbf{i}x)/\mathbf{i} = \frac{2}{\sqrt{\pi}} \int_0^x e^{u^2} du$. Although the relation contains $\mathbf{i}$, both $\mathrm{erf}(x)$ and $\mathrm{erfi}(x)$ are real functions. So for the case when $\tau_\mathrm{b} < \tau_\mathrm{f}$, we get the modified real equation

$$\begin{aligned} I(D) = I_0 e^{-D\frac{\tau_\mathrm{f}+\tau_\mathrm{b}}{2}} + L_\mathrm{f} - L_\mathrm{b} e^{-D\frac{\tau_\mathrm{f}+\tau_\mathrm{b}}{2}} \\ + (L_\mathrm{b} - L_\mathrm{f}) \frac{1}{\sqrt{D(\tau_\mathrm{f}-\tau_\mathrm{b})}} e^{-\frac{D}{2(\tau_\mathrm{f}-\tau_\mathrm{b})}\tau_\mathrm{f}^2} \sqrt{\frac{\pi}{2}} \\ \left[\mathrm{erfi}\left(\tau_\mathrm{f} \frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}} \right) - \mathrm{erfi}\left(\tau_\mathrm{b} \frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}} \right) \right] \end{aligned} \quad (2.22)$$

Neither Equation 2.21 nor Equation 2.22 is well defined if $\tau_\mathrm{b} = \tau_\mathrm{f}$. For the special case when the attenuation coefficient is constant, we have to resolve Equation 2.2 and get yet another equation:

$$I(D) = I_0 e^{-\tau D} + L_\mathrm{b}\left(\frac{1}{\tau D} - \frac{1}{\tau D} e^{-\tau D} - e^{-\tau D} \right) + L_\mathrm{f}\left(1 + \frac{1}{\tau D} e^{-\tau D} - \frac{1}{\tau D} \right) \quad (2.23)$$

Equations 2.21 through 2.23 together make up the closed form for the volume rendering integral with linear parameters.

Chapter 3

Practical Implementations of Volume Rendering

In the previous chapter, I introduced the volume rendering integral, which has provided the foundation of volume rendering for scientific visualization since the late 1980s. In this chapter, I will review the current techniques used to apply the volume rendering integral to volumetric models.

We can break the process of volume rendering into two tasks. The first task is that of determining **cell-ray intersections**. This task is the process of determining which cells each viewing ray intersects. The result is a list (or stream) of samples (or segments) along each viewing ray enumerating the properties of the volume.

The second task is that of performing **color computations**. This task is the process of applying the volume rendering integral to the properties of the volume already sampled along the ray.

Although I describe volume rendering as a two-step process, in reality, volume renderers most often perform these two tasks together in a pipeline configuration.

Even if we do not perform the process in a true hardware pipeline, pipelining the tasks prevents the system from having to store the intermediate samples along viewing rays, which are numerous for high quality renderings.

3.1 Cell-Ray Intersections

In this section, I discuss the process of determining how the cells in a volumetric model project onto a viewing screen. As discussed in Section 1.2, in a graphics hardware pipeline, this process is the geometric processing. Consequently, when implementing volume rendering on graphics hardware, we usually perform cell-ray intersections on the vertex processor.

In the first two sections, I discuss general methods that work on unstructured grids, which are the types of models this dissertation is mostly concerned. For completeness, I discuss also techniques for rendering regular grids. However, I do not expand on regular grid rendering as the problem of cell-ray intersections is simpler than that for unstructured grids and errors in color computations are less noticeable. Moreover, there already has been significantly more research on the rendering of regular grids. This is because there is a large amount of models defined as regular grids, particularly in medical visualization where CT, MRI, and ultrasound scans result in a regular grid of samples. Nevertheless, unstructured grids are an important modeling tool that can provide far more accuracy with many fewer data. For example, Leven and colleagues [58] built a volume renderer that resampled unstructured grids with regular grids. To create enough samples to maintain the accuracy of the unstructured grids, their data could grow by three orders of magnitude.

3.1.1 Ray Casting

When volume rendering with **ray casting** (known also as **ray tracing**), the system traces rays from the viewpoint through each pixel of the image and determines what cells each ray intersects. If we desire global effects such as shadows or scattering, then we can trace also other rays from cell intersections to light sources or other cells (although computational cost quickly becomes prohibitive). Because the rendering system, in its outer loop, iterates over all the pixels in the output image, ray casting is known also as **image-order rendering**.

Naïve Ray Casting

The basic operation of a ray caster is, given a ray originating at a point, determining the first object that the ray intersects. Determining where a ray intersects a cell is a simple and quick operation for practical cells such as polyhedra [1]. To determine the first cell a ray intersects, we could simply intersect the ray with every cell and pick the intersection closest to the origination point in the direction of the ray.

Although this approach works, it is unnecessarily computationally intensive to intersect a ray with every cell, and the first-intersection operation must occur many times during a render operation. Therefore, any practical ray casting renderer arranges the objects in a spatial hierarchy to prune away cells quickly as shown in Figure 3.1. The pruning hierarchy is sufficient for ray tracers that principally render opaque surfaces [1]. However, rays intersect far too many cells of a translucent volume (and therefore the system must perform too many first-intersection operations) to make this direct approach practical for volume rendering.

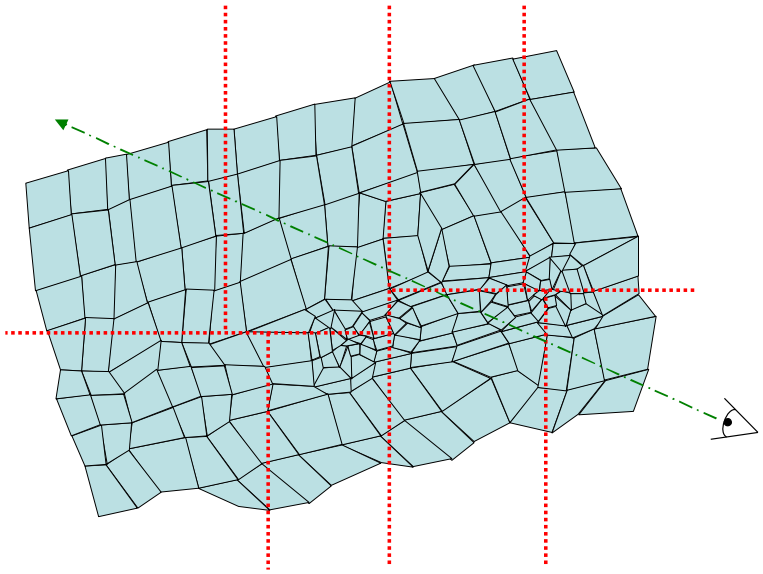
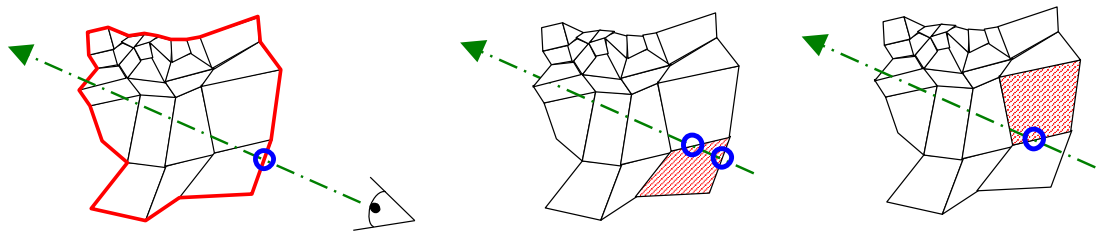


Figure 3.1: An example of hierarchical pruning for ray casting (in 2D). With a K-D tree structure, the application can isolate the nearest region in $O(\log(N))$ time.

Cell Traversal

Garrity [31] introduces an efficient method for ray casting transparent unstructured grids. His method requires that the unstructured grid be a **conforming mesh**. For a mesh to be conforming, the intersection of any two of its cells is either empty or the shared vertex, edge, or face. Non-conforming grids hold a variety of problems with both simulation and visualization and are therefore usually considered to be erroneous grids.

We can partition the cell faces in a conforming unstructured mesh into two sets. The **internal faces** are those that are shared by two cells. The **external faces** (known also as **boundary faces**) are those that belong to only one cell. Garrity's method starts with the obvious observation that a ray originating from outside a grid must enter it through an external face. In general, the number of external faces is drastically smaller than the total number of cells. Thus, checking only the external



(a) A ray intersecting an external face.

(b) The ray exiting the cell.

(c) The ray entrance to the next cell.

Figure 3.2: Ray tracing with cell traversal. A ray originating outside the grid first intersects an external face (a). Once inside a cell, a ray must exit another face of the cell (b). Once we determine the exit face, we can retrieve the next cell from the cell connectivity graph (c).

faces drastically reduces the number of intersection tests. Bunyk, Kaufman, and Silva [5] speed up this part of the algorithm further by computing all ray intersections of each front facing external face at one time.

Once inside the cell, the ray, obviously, must exit through one of its faces. Thus, at this point we need only to check intersections of the ray with the cell's faces. If the exit face happens to be an external face, then we can find the next entry point by again checking only external faces. If the exit face is an internal face, then, by definition, another cell shares the face. These shared faces are static with respect to the grid. We could therefore generate offline a **cell connectivity graph**, a graph where each node represents a cell and each edge represents a face shared between cells. We could then consult the cell connectivity graph to determine the next cell without further intersection tests. Figure 3.2 demonstrates this process.

Weiler and colleagues [99] developed a way to perform ray casting using cell traversal on graphics hardware. Weiler finds the initial ray entry point by rendering front faces. Weiler then traverses through cells using the fragment program by storing the cells and connectivity graph in textures. Once a ray exits the grid, Weiler's

algorithm does not handle reentry into the grid. Thus, the grids must be convex. To handle nonconvex grids, Weiler adds cells to make the grid convex in the same manner as suggested by [103].

Sweeping

The major inefficiency with the previously introduced ray casting algorithms is that they do not take advantage of coherency. The viewing rays from two adjoining pixels are likely to intersect many of the same cells in the model. However, the previous ray casting algorithms independently recast every ray, duplicating much of the work.

One way to take advantage of ray coherency is with a **sweep** algorithm. Sweeping is a general-purpose idea in computational geometry that helps reduce the dimensionality of the problem [17].

Giertsen [32] was the first to apply sweeping to volume rendering unstructured grids. Giertsen placed a **sweep plane** perpendicular to the view plane at an extreme point of the model. Giertsen then swept the plane through the model. As the sweep plane passed through them model, Giertsen would maintain the set of polygons formed by the intersection of the model and the sweep plane.

The sweep algorithm works because incrementally updating the intersection is much easier than cutting the model by an arbitrary plane. As with all sweep algorithms, the sweep plane in principle moves continuously, but in practice, the sweep algorithm processes the plane only at certain events. In Giertsen's case, the events are the vertices of the model, where the topography of the polygons changes, and the scan lines, where rays are cast through the plane to determine pixel colors. Silva [87, 88, 89] made several improvements to Giertsen's algorithm. The most notable change is the use of a **sweep line** to determine the intersections of polygons in the plane with viewing rays.

Yagel and colleagues [111] introduce another sweep plane algorithm. Unlike Giertsen, Yagel uses a sweep plane parallel to the view plane. Yagel then renders the polygons in the sweep plane directly on graphics hardware, making the algorithm, in practice, much faster than Giertsen's. However, Yagel's sweep plane stops only at a predetermined number of locations making it possible (even likely) that the sweep plane completely misses cells. Weiler and Ertl [97] show how to use multitextures to perform both the slicing of the cells in addition to rasterizing the resulting polygon.

3.1.2 Cell Projection

When volume rendering with **cell projection**, the system traverses the list of cells and projects each one onto the viewing plane. Once the cell's projection is determined, the renderer fills the pixels covering that part of the viewing plane, a process we call **rasterization**. Because the rendering system, in its outer loop, iterates over all the objects, cell projection is known also as **object-order rendering**.

The major advantage cell projection has over ray casting is that every viewing-ray intersection with a cell is computed at once when the cell is projected, making it easy to take advantage of coherency among viewing rays without the overhead of a sweep algorithm. Furthermore, because commodity graphics hardware is also an object-order rendering system, it is straightforward to implement the cell projection algorithms on graphics hardware, making them faster than CPU-bound algorithms. The major disadvantage of cell projection is its reliance on proper visibility ordering of the cells. Therefore, in addition to reviewing the most popular cell projection algorithms, I review common cell sorting algorithms.

Projected Tetrahedra

Shirley and Tuchman [86] proposed the first algorithm for projecting cells of unstructured grids. Like most other cell projection algorithms, Shirley and Tuchman's algorithm, for simplicity, works with a single type of polyhedron: the tetrahedron. Because their algorithm works exclusively with tetrahedra, they dubbed their algorithm **projected tetrahedra**. Thanks to its simplicity and effectiveness, a significant number of rendering systems still uses the projected tetrahedra algorithm today.

Shirley and Tuchman chose the tetrahedron because it is a **simplex** in three dimensions [36]. That is, the tetrahedron is the simplest possible polyhedron; it has the minimum number of vertices (4), edges (6), or faces (4) required to construct a polyhedron. A tetrahedron is always **simple** (non self-intersecting) and **convex** (any segment connecting two points within the tetrahedron is completely contained by the tetrahedron). Furthermore, we can decompose any simple polyhedron into tetrahedra. Thus, the projected tetrahedra algorithm will work for general unstructured grids once we decompose them into tetrahedra.

Shirley and Tuchman noted that when a tetrahedron projects onto a viewing plane, they could classify it in one of four ways, shown in Figure 3.3. If the tetrahedron is in general position, it will fall into Class 1 or Class 2. If one or two faces are perpendicular to the viewing plane, it will fall into Class 3 or Class 4, respectively. By comparing the dot products of the surface normals with the viewing vector, Shirley and Tuchman were able to classify the tetrahedra.

Once it has determined the projection class, the algorithm can break the projection into triangles. Figure 3.3 shows how Shirley and Tuchman decompose each projection into triangles. No triangular region crosses an edge of the projected tetrahedra. Thus, assuming we interpolate parameters linearly through the tetrahedron,

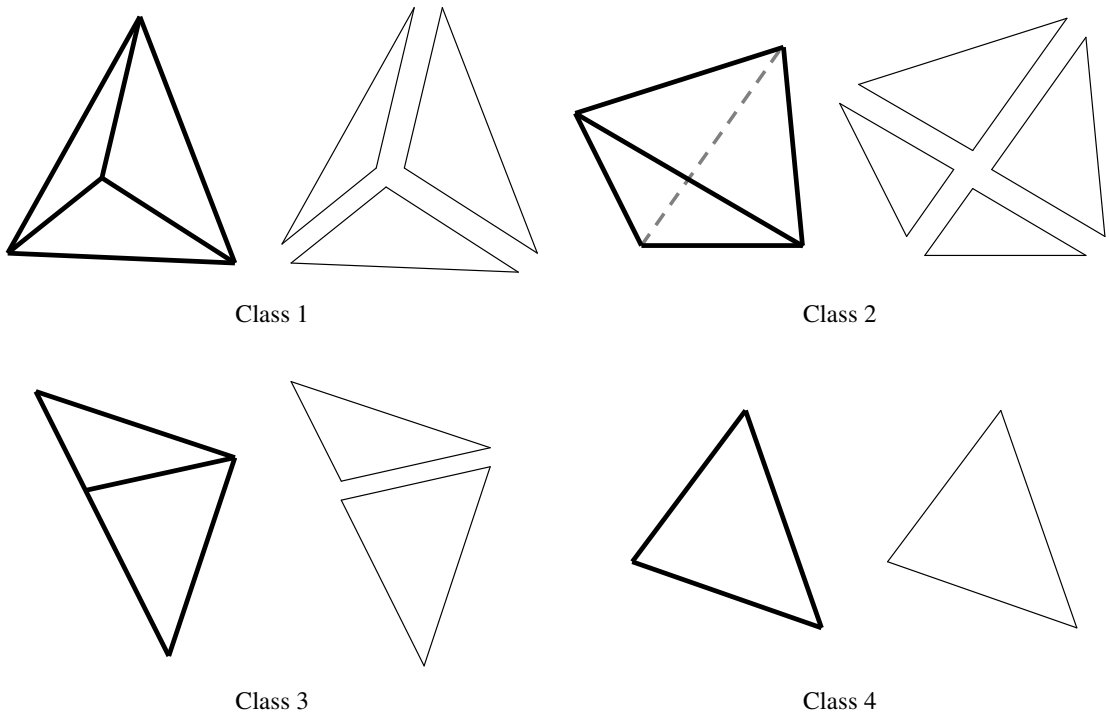


Figure 3.3: Projected tetrahedra classes. The four cases comprise the possible ways a tetrahedron projects onto a 2D viewing plane. For each case, this figure demonstrates how we can decompose the tetrahedron into triangles.

the parameters vary linearly within the triangles. Once the triangles have been determined, Shirley and Tuchman feed them to graphics hardware to render.

Wilhelms and van Gelder [102] propose a similar algorithm that projects hexahedra instead of tetrahedra. Although the hexahedron is not as versatile as a tetrahedron—in general, a polyhedron cannot be decomposed into hexahedra—it is a common element in unstructured grids. Furthermore, we require five or six tetrahedra (depending on layout) to decompose a hexahedron. Therefore, projecting the hexahedra directly can lead to a substantial performance improvement.

GPU Accelerated Projected Tetrahedra

Although hardware accelerated, the projected tetrahedra algorithm requires a substantial amount of CPU usage. Ideally, we would like our graphics hardware to project polyhedra (or at least tetrahedra) in the same manner as it projects points, lines, and polygons. King, Wittenbrink, and Wolters [45] propose an architecture to do just this, but neither they nor anyone else, have implemented their architecture.

Once programmable graphics hardware became available, Wylie and colleagues [110] devised a means of performing tetrahedra projection completely on the graphics card. They called their method the GPU Accelerated Tetrahedra Renderer (**GATOR**).

The limitations of vertex programs were Wylie's biggest challenge. At the time, vertex programs had neither the ability to branch nor the ability to add or subtract vertices.[1] Therefore, Wylie had to know the number of triangles to use *a-priori*.

To get around this problem, Wylie builds a **basis graph**, shown in Figure 3.4. If we treat each projected tetrahedra projection class (shown in Figure 3.3 on page 37) as a graph, we notice that they are all isomorphic with the basis graph (assuming we allow nodes of the basis graph to be located at a single point). Wylie draws the basis graph as a triangle fan, so the problem reduces to finding a mapping from a projection to the basis graph.

To find this mapping, Wylie enumerates all the permutations of the projections, shown in Figure 3.5. There are fourteen permutations in all, which GATOR uniquely identifies with four tests. Three of the tests involve checking the direction of the cross product of various vectors along edges. The fourth test involves checking whether two particular segments intersect. Once GATOR properly identifies the permutation,

[1] As of the time of the writing of this dissertation, there is still no way to change cell topology from within a vertex program.

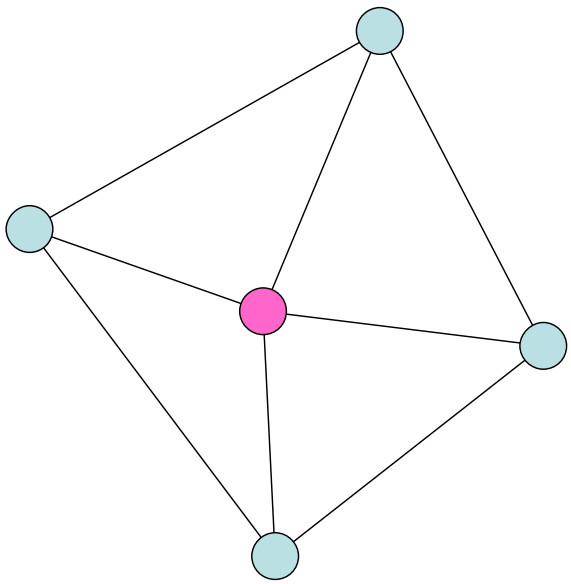


Figure 3.4: GATOR basis graph.

GATOR retrieves the appropriate mapping of the basis graph via a lookup table.

View Independent Cell Projection

Weiler, Kraus, and Ertl [98] developed another method for performing cell projection completely within a graphics card that deviates significantly from the projected tetrahedra method. They call their algorithm **view independent cell projection** because, unlike projected tetrahedra, the process does not change with the viewing position.

When rasterizing a projected cell, any cell projection algorithm must calculate two intersections per pixel: the entry and exit points of the viewing ray. Finding one of these two intersections on graphics hardware is trivial: simply rasterize the polygon face. Doing this gives either the entry point (if it is a front-facing polygon) or the exit point (if it is a back-facing polygon).

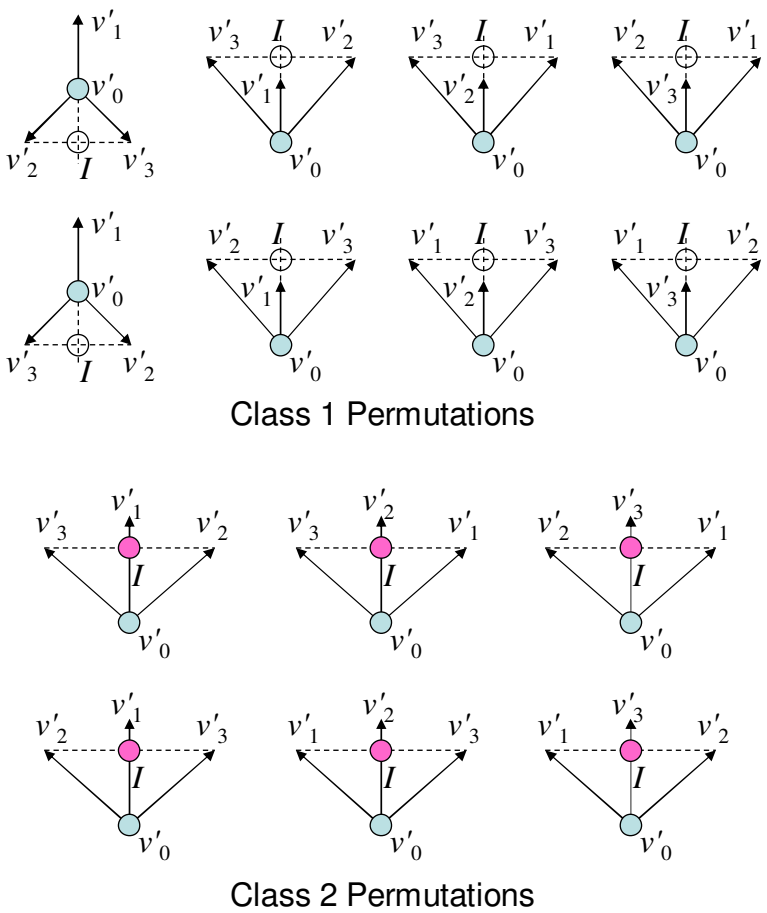


Figure 3.5: All the permutations of the first two classes of the projected tetrahedra algorithm. Each permutation has a unique mapping to the basis graph (given in Figure 3.4 on page 39).

Weiler, Kraus, and Ertl's algorithm proceeds by using the graphics hardware rasterizer to find all the viewing ray entry points. They find all the entry points by drawing the front faces of the tetrahedra. They then use the fragment processor to determine the exit point for each viewing ray. They do not need to rasterize the back faces, so the algorithm culls these faces.[2]

To find the exit point, Weiler, Kraus, and Ertl intersect the viewing ray with each

[2]Culling back facing polygons is a standard OpenGL operation[109].

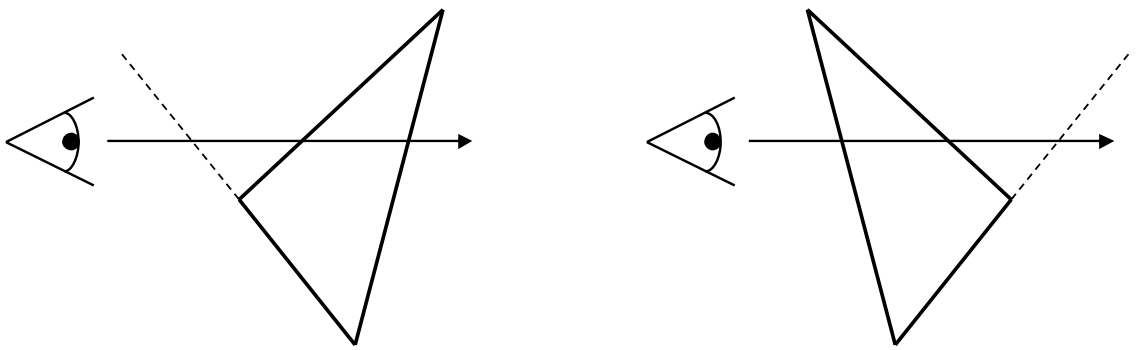


Figure 3.6: 2D example of the intersection of viewing rays with the planes of cell faces. If a ray does not intersect a cell face, then it intersects the plane containing that face either before the entrance (shown on the left) or after the exit (shown on the right).

plane containing a face. Unless a face is parallel to the viewing ray, every plane will intersect this viewing ray. However, as demonstrated in Figure 3.6, the intersection of the ray and any plane containing a face not intersected by the ray will occur either before the entry point or after the exit point [34]. Therefore, the true exit point is the one closest to the entry point that is not before the entry.

Finding the intersection of a plane and a ray is easy. Consider the implicit equation for a plane:

$$\vec{n} \cdot \vec{x} + a = 0 \tag{3.1}$$

where $\vec{n}$ is the normal to the plane. Finding the plane equation for a face is as simple as finding the face's normal and then plugging in a known point in the face (*i.e.* one of its vertices) into $\vec{x}$ to find a. For the ray, we use the parametric equation

$$\vec{r} = \vec{v} + t\vec{d} \tag{3.2}$$

where $\vec{v}$ is a point on the ray and $\vec{d}$ is a vector pointing in the direction of the ray. If $\vec{v}$ is the entry point of the ray into the tetrahedron (which front-face rasterization gives) and $\vec{d}$ is the normalized viewing ray, then t is the distance between the entry

point and the intersection of the plane. Negative values for t mean the intersection occurs in front of the entry point.

We can find the intersection by plugging Equation 3.1 into Equation 3.2 and solving for the only unknown, t.

$$\begin{aligned}
\vec{n} \cdot \left(\vec{v} + t\vec{d}\right) + a &= 0 \\
\vec{n} \cdot \vec{v} + t\vec{n} \cdot \vec{d} + a &= 0 \\
t\vec{n} \cdot \vec{d} &= -\left(\vec{n} \cdot \vec{v} + a\right) \\
t &= -\frac{\vec{n} \cdot \vec{v} + a}{\vec{n} \cdot \vec{d}}
\end{aligned} \tag{3.3}$$

The actual exit intersection is that with the smallest value of t that is greater than zero.

Once they determine the back intersection point, Weiler, Kraus, and Ertl still need to know the scalar value at the back intersection point. Assuming that the gradient, $\vec{g}$, of the scalar is constant (and known), the scalar value at the back is

$$s_b = s_f + \left(\vec{d} \cdot \vec{g}\right) D \tag{3.4}$$

where s_b and s_f are the values of the scalar at the front and back, respectively, of the ray, $\vec{d}$ is again the normalized viewing ray, and D is the distance between the front and back intersections (equal to t in Equation 3.3).

Given scalar values at the vertices of the tetrahedron, there exists a unique, consistent, and constant gradient. We can find this gradient by establishing a system of equations applying Equation 3.4 to all the edges connected to one vertex.

$$\begin{aligned}
s_1 &= s_0 + (\vec{v_1} - \vec{v_0}) \cdot \vec{g} \\
s_2 &= s_0 + (\vec{v_2} - \vec{v_0}) \cdot \vec{g} \\
s_3 &= s_0 + (\vec{v_3} - \vec{v_0}) \cdot \vec{g}
\end{aligned}$$

We can change these equations into a more familiar matrix form.

$$\begin{bmatrix} s_1 - s_0 \\ s_2 - s_0 \\ s_3 - s_0 \end{bmatrix} = \begin{bmatrix} (\vec{v_1} - \vec{v_0})^{\mathrm{T}} \\ (\vec{v_2} - \vec{v_0})^{\mathrm{T}} \\ (\vec{v_3} - \vec{v_0})^{\mathrm{T}} \end{bmatrix} \vec{g} \tag{3.5}$$

We can solve Equation 3.5 with elementary linear algebra [2]. Note that the gradients do not change with the viewpoint, and they can therefore be determined offline.

At the time Weiler, Kraus, and Ertl were developing their algorithm, the fragment processor of commodity graphics cards was not powerful enough to perform the ray-plane intersections, scalar determination, and ray integration.[3] However, they observe that the distance from a face to a plane and the scalar value on the opposite plane varied linearly across a face of a tetrahedron. As such, they can compute the ray-plane intersections and scalar values at the vertices in a vertex program. They then store the depths and scalars in triples as texture coordinates that the rasterizer linearly interpolates.

Weiler, Kraus, and Ertl note another advantage of performing ray-plane intersections at cell vertices. At each vertex, the algorithm performs three intersection calculations. However, as demonstrated in Figure 3.7, each vertex belongs to two of the three faces with which they perform intersection calculations. Because the vertex is part of these planes, the intersection at these planes is at the vertex. Thus, the algorithm does not need to solve the intersection with these planes explicitly.

Visibility Sorting

When performing ray casting, we implicitly know the order in which a ray intersects cells. However, when performing cell projection using any of the previously discussed

[3] Weiler and colleagues [100] later demonstrated how to perform all these operations on the next generation of graphics cards.

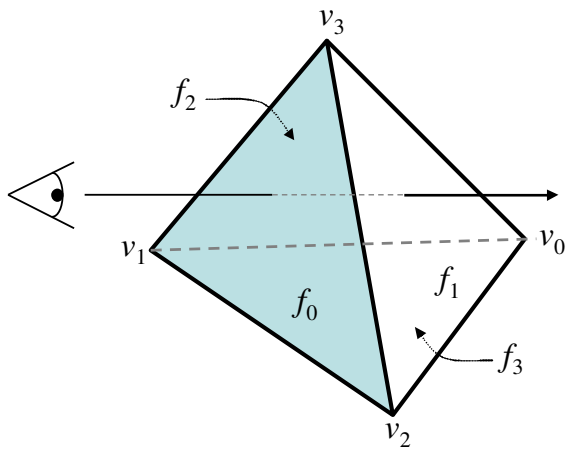


Figure 3.7: Ray-plane intersections at vertices. When rasterizing face f_0, the distance to the planes of faces f_1, f_2, and f_3 must be computed. However, note that, for example, vertex v_1 touches f_2 and f_3. Thus, the distance to those faces is zero and the scalar for that intersection is that at v_1.

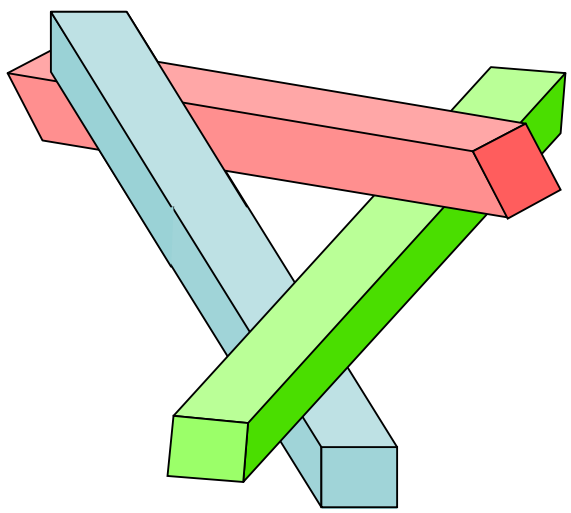


Figure 3.8: A visibility cycle.

algorithms, we must first determine the visibility order so that we can project the cells in the proper order.

Formally, the **visibility-sorting** problem is that of finding a relationship among

cells for a given viewpoint. We define the relationship $\prec_{\mathrm{v}}$ such that if any part of cell C_i occludes any part of cell C_j, then $C_i \prec_{\mathrm{v}} C_j$. We require our visibility-sorting algorithm to build an array $A = \{a_1, a_2, \ldots, a_n\}$ such that $\forall i, j : a_i \prec_{\mathrm{v}} a_j \rightarrow i \leq j$. There is no guarantee that such an array will exist. Figure 3.8 demonstrates how as little as three convex cells can be mutually occluding. We call this mutual occlusion a **visibility cycle**.

When faced with a visibility cycle, a visibility-sorting algorithm has only two choices. The first choice is the **ostrich algorithm**: ignore the problem and hope that visibility cycles either do not happen or do not make noticeable visual artifacts. The second choice is to split cells so that the cycle is broken.

Newell, Newell, and Sancha [70] developed a direct algorithm for performing visibility sorting of polygons in the early 1970's. Their algorithm relies on a routine that determined whether a cell C_i occluded a cell C_j. To speed up this operation, Newell, Newell, and Sancha use a sequence of operations increasing in accuracy and complexity to determine the truth of the relationship. Because the $\prec_{\mathrm{v}}$ relationship is not transitive, we cannot use this operation as the comparator in a standard $O(n \log n)$ sorting algorithm. Instead, Newell, Newell, and Sancha order the cells based on their distance from the viewpoint and then compare only pairs of cells whose depth ranges overlap. In principle, this could lead to $O(n^2)$ algorithmic behavior, but in practice, far fewer comparisons are performed. Stein, Becker, and Max [92] extend the comparison operation to work with polyhedra.[4]

Other methods of sorting polygonal cells do not extend to polyhedrons. The **binary space partitioning (BSP) trees** algorithm [29, 30] uses cutting planes to divide space into a binary tree that the algorithm can quickly walk to determine a visibility order. However, when applied to meshed polyhedra, the number of cells

[4]The comparison operation in Stein, Becker, and Max has a bug that is fixed by Williams, Max, and Stein [105].

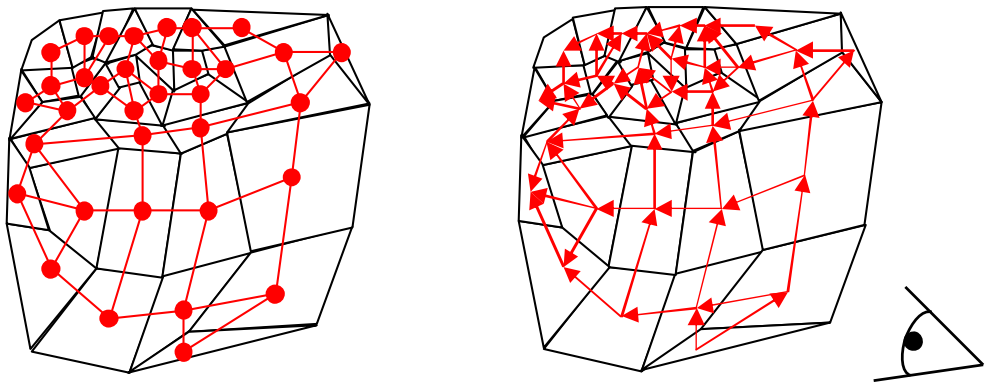


Figure 3.9: An example of an MPVO cell connectivity graph. On the left is a convex mesh with its connectivity graph superimposed. On the right, a viewpoint imposes directionality on the edges.

can grow quadratically [72]. deBerg, Overmars, and Schwartzkopf [15, 16] present a quicksort-like algorithm that used the transitive closure of $\prec_v$ as a comparison operator. However, their method of determining if a pair of cells exists in this transitive closure does not extend to polyhedra.

Williams [103] presents the well-known **Meshed Polyhedra Visibility Ordering (MPVO)** algorithm.[5] Williams' MPVO algorithm requires several restrictions on the mesh. First, the mesh has to be **fully connected**. A connected mesh is in one piece. More formally, you cannot partition the cells of a connected mesh into two sets such that the partitions share no faces. Second, the mesh has to be **convex**. A convex mesh contains all the faces of the **convex hull** of its vertices. A convex hull of a set of points (in Euclidean three space) is the smallest polyhedron containing all the points. Third, the mesh must have no **holes**, meaning the mesh cannot enclose any empty regions. These restrictions ultimately mean that a ray exiting the mesh cannot reenter the mesh.

The MPVO algorithm first preprocesses the mesh by building a connectivity

[5] Max, Hanrahan, and Crawfis [64] independently developed a similar algorithm.

graph. The connectivity graph represents each cell with a node and each shared face with an edge. Given a viewpoint for visibility sorting, the MPVO algorithm gives directionality to each edge based on which side of the associated face is facing the viewpoint. The cell on the front face occludes the cell at the back face. Assuming no visibility cycles are present, adding edge directionality results in a **directed acyclic graph** (**DAG**). The DAG thus reduces sorting to walking the tree.

The biggest problem with MPVO is its reliance on connected, convex meshes, a requirement seldom enforced in unstructured grids. To deal with nonconvex meshes, Williams [103] proposes also an extension to MPVO called **MPVONC**. The MPVONC extension provides several heuristics for choosing a total ordering from the partial ordering of the DAG that is likely to be correct for occlusions outside of the DAG. However, there is no guarantee that these heuristics will be correct. Williams [103] suggests also filling concavities and holes with dummy cells used for the sorting but not for the drawing.

Silva, Mitchell, and Williams [90] extend the MPVO algorithm with the **XMPVO** algorithm. XMPVO works exactly like MPVO except that at every viewpoint a sweep plane algorithm determines any occlusions caused by rays exiting and then reentering the mesh. XMPVO adds these occlusions as edges to the DAG. Comba and colleagues [11] again improve the algorithm with **BSP-XMPVO**. BSP-XMPVO is just like its predecessor except that it uses a BSP tree to determine occlusions instead of a sweep plane. The BSP tree requires more preprocessing and more memory, but is ultimately faster to compute per viewpoint change than the sweep plane algorithm.

For the special case of **Delaunay triangulations**, Max, Hanrahan, and Crawfis [64] give a simple method for visibility sorting. A Delaunay triangulation (in 3D) is one that has the property that the circumsphere of the vertices of each tetrahedron contains no other vertices [17].

Let the **power distance** of a point with respect to a sphere be $d^2 - r^2$, where d is the distance between the point and the sphere's center, and r is the radius of the sphere. Max, Hanrahan, and Crawfis show that the sorting of the power distances between the viewpoint and the circumspheres of the tetrahedra in a Delaunay triangulation is equivalent to the visibility sorting of the tetrahedra. However, the method is valid only for proper Delaunay triangulations and is sensitive to degenerate cases. Cignoni and De Floriani [9] provide an algorithm to create valid sets of spheres for general meshes, but their algorithm is complex and not guaranteed to work for all meshes.

Algorithms exist for detecting and projecting visibility cycles. Snyder and Lengyel [91] extend the Newell, Newell, and Sancha algorithm to detect cycles. Kraus and Ertl [52] extend MPVO to detect cycles, making **MPVOC**. Both algorithms use an image based approach to project cells with a visibility cycle correctly rather than split the cells geometrically.

Ideally, graphics hardware would employ an image-based ordering solution much like the **z-buffer algorithm** for opaque surfaces [8]. However, the z-buffer algorithm gives only the closest surface where volume rendering requires the ordering of every cell projected on a pixel. Carpenter [7] proposes the **A-buffer**. The A-buffer maintains a linked list of depth sorted fragments at every pixel. However, such data structures are difficult and slow to implement on graphics hardware. Given enough oversampling of the image, one could implement **screen door transparency**, which renders pixels opaque but writes only some of the pixels based on the opacity [28, 69]. Jouppi and Chang [39] propose an improved screen door transparency called Z^3. However, Jouppi and Chang demonstrate accuracy to depth complexities of only 16, which is nowhere near deep enough for volume rendering. Wittenbrink [107] proposes the **R-buffer**, which serves the same function as the A-buffer but stores fragments in a single array rather than a collection of linked lists. Although more practical than

the A-buffer, it still has yet to be implemented in graphics hardware. Furthermore, a 1000 by 1000 pixel volume rendering with a modest average depth complexity of 100 requires over 600 megabytes of storage.

ZSWEEP

Farias, Mitchell, and Silva [24, 25] propose the **ZSWEEP** algorithm. Although it has its roots in sweep plane algorithms, ZSWEEP actually projects cells onto the viewing plane. The basic idea behind ZSWEEP is simple. First, ZSWEEP sorts the vertices of the mesh based on the current viewpoint. The algorithm then visits the vertices one at a time in back to front order. When it visits a vertex, ZSWEEP projects all faces attached to that vertex that are not yet projected. Using information stored in frame buffers, ZSWEEP can extract the parameters necessary for volume integration.

Of course, sorting vertex depths alone is not sufficient to create a proper depth ordering of the cells. However, tests performed by Farias, Mitchell, and Silva show that about 70% of the fragments projected are in front of all those previously projected, about 82% are behind no more than one other fragment, and over 99% are behind no more than two. Farias, Mitchell, and Silva conclude that, given their order of projection, they need in practice an A-buffer of only limited depth. Although A-buffers of even limited depth are not available yet on commodity graphics hardware, one could be implemented with high efficiency. Furthermore, ZSWEEP stands alone in accuracy and speed for software implementations of unstructured grid volume rendering.

3.1.3 Rectilinear Grid Resampling

Volume rendering rectilinear grids is significantly easier than for unstructured grids. For example, tracing a ray through a rectilinear grid is a simple operation and would

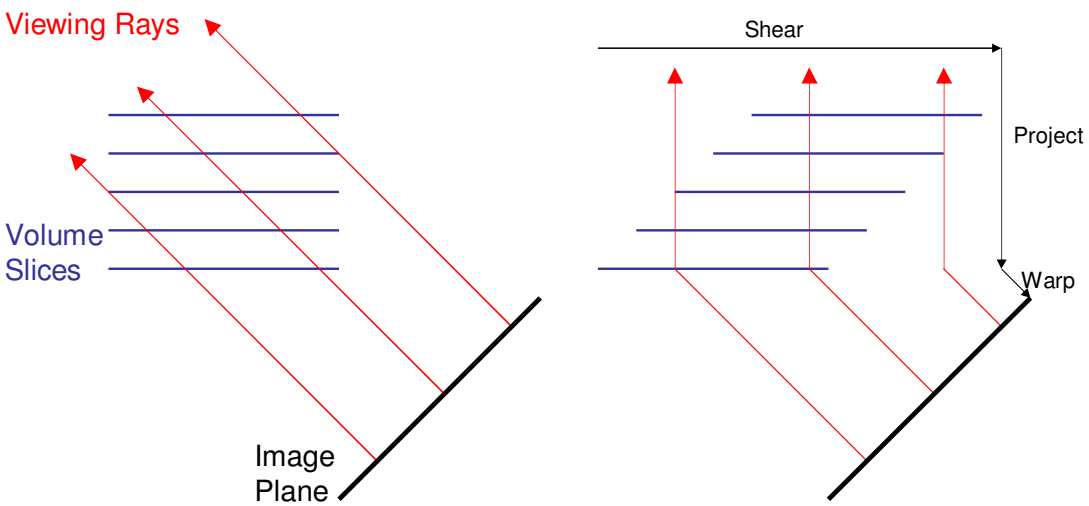


Figure 3.10: Shear-warp factorization. The image on the left shows the traditional casting of rays. The image on the right shows a progression of shear, project, and then warp transformations that provide the equivalent ray-cell intersections.

make for a reasonable, if naïve, ray caster. However, this type of implementation is not optimal. For example, as the ray traverses the volume it skips over large portions of data in memory, usually killing cache performance. Furthermore, nearby rays may touch the same elements in the rectilinear grid, which causes the same values to be loaded multiple times.

Lacroute and Levoy [56] provide a way to resample a rectilinear volume that is significantly faster than the naïve approach. They call their method **shear-warp factorization**. The shear-warp factorization algorithm's ability to traverse the volume memory consecutively helps make it the fastest known algorithm for CPU-based volume rendering.[6]

Rather than trace rays through the volume, Lacroute and Levoy instead traverse the volume data in the order it is stored in memory, slice by slice. They warp the

[6]According to Pfister and colleagues [73] based on experiments performed by Lacroute [55].

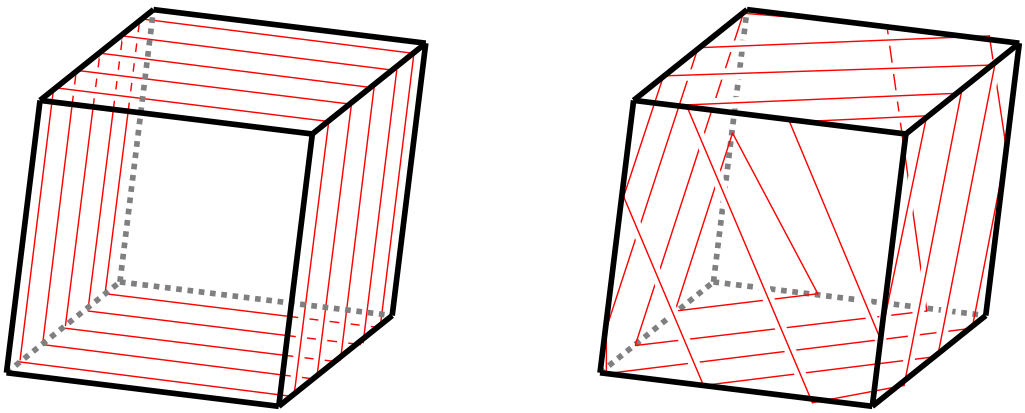


Figure 3.11: Potential slicing of rectilinear volumes for texture-based volume rendering. On the left are object-aligned slices. On the right are view-aligned slices.

slices to project properly on the image plane. As the algorithm moves from one slice to the next, it shears the slices of the volume so that the same warp may be used. Figure 3.10 demonstrates how this approach is equivalent to casting rays.

We can use commodity graphics hardware also to render rectilinear volumes effectively. The most successful approaches simply render slices of the volume as polygons while using the texture hardware to map the data onto the slices. Only the speed at which fragments may be processed and the amount of texture memory available limit the method. We can obtain greater speeds by using fewer slices, but this reduces the sampling of the grid and, therefore, aliasing quickly becomes a problem.

When using texture hardware to render rectilinear volumes, the system may slice the volume in one of two ways [12], as shown in Figure 3.11. The first mode of slicing is **object-aligned slicing**. With object-aligned slicing, the slices are fixed to the volume, much like in shear-warp factorization. Because the slices are fixed with respect to the volume, the data for each slice may be stored in a 2D texture. Of course, slices will not be visible if they are parallel to the viewing rays. For object-aligned slices to work, there must be at least three copies of the slices where

each set is perpendicular to a principle axis of the volume. Furthermore, as the viewing rays deviate from the principle axes, the spacing and interpolation between the slices changes, leading to errors. Recent advances in graphics hardware allow us to compensate for these errors [79].

The second mode of slicing is **view-aligned slicing**. With view-aligned slicing, the slices are always perpendicular to the view plane and the renderer trilinearly interpolates the volume data to map onto the slice. Cabral, Cam, and Foran [6] perform this interpolation on the CPU [6], but a more efficient approach is to use 3D textures [106].

A significant problem with using texture hardware on a traditional OpenGL pipeline is that the texture holds the final colors. This means that classification and shading cannot change without reloading all the texture data. However, recent research and hardware improvements have largely corrected this problem [13, 22, 68, 79, 95, 101].

3.2 Color Computations

In this section, I discuss the process of taking samples of volume material properties along a viewing ray and applying the volume rendering integral to determine the light intensity at each pixel. As discussed in Section 1.2, in a graphics hardware pipeline, we refer to this process as fragment processing. Consequently, when implementing volume rendering on graphics hardware, we perform color computations on the fragment processor.

Ultimately, the goal of the color computations is to compute the volume rendering integral, defined as

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \tag{3.6}$$

I derive and discuss Equation 3.6 in detail in Chapter 2. In this section, I discuss practical methods of evaluating this integral.

3.2.1 Riemann Sum

A simple (albeit inaccurate) method for numerical approximations of integrals is the **Riemann sum**. The idea behind the Riemann sum is to take an integral, for example of the form

$$\int_a^b f(x)dx \tag{3.7}$$

and break it up into a finite amount of pieces that we sum together. The sum is

$$\sum_{i=1}^{n} f(x_i)\Delta x \tag{3.8}$$

where $\Delta x = (b-a)/n$ and the sample x_is are chosen such that $(i-1)\Delta x \leq x_i \leq i\Delta x$. As Δs approaches zero (and n approaches infinity), Equation 3.8 converges to Equation 3.7. Consequently, using more terms in the sum results in a more accurate numerical approximation.

Max [63] gives an overview of how we may apply the Riemann sum to the volume rendering integral. The volume rendering integral (Equation 3.6) actually has several integrals in it. The first is the integral attenuating the incoming light. We approximate $\exp\left(-\int_0^D \tau(t)dt\right)$ as follows.

$$\exp\left(\sum_{i=1}^{n}\tau(t_i)\Delta t\right) = \prod_{i=0}^{n}\exp\left(\tau(t_i)\Delta t\right) = \prod_{i=0}^{n}\zeta_i \tag{3.9}$$

Note that in Equation 3.9 I have substituted ζ_i for $\exp\left(\tau(t_i)\Delta t\right)$. Assuming that Δt is fixed, we can precompute values of ζ_i based on the associated values for $\tau(t_i)$.

We approximate the second outer integral of Equation 3.6 in much the same way. $\int_0^D L(s)\tau(s)\exp\left(\int_s^D \tau(t)dt\right)$ becomes

$$\sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} \zeta_j \tag{3.10}$$

Putting Equation 3.9 and Equation 3.10 together, we get the following approximation for the volume rendering integral.

$$\begin{aligned} I(D) &\approx I_0 \prod_{i=0}^{n} \zeta_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} \zeta_j \\ &= g_n + \zeta_n(g_{n-1} + \zeta_{n-1}(g_{n-2} + \zeta_{n-2}(g_{n-3} + \cdots \zeta_2(g_1 + \zeta_1 I_0)\cdots))) \end{aligned} \tag{3.11}$$

Equation 3.11 yields simple front-to-back or back-to-front methods for computing it.

Until now, I have been intentionally vague on the form of g_i. Max [63] defines it as the traditional Riemann sum form dictates it: $g_i = L(s_i)\tau(s_i)\Delta s$. However, in Section 2.3.3, I show that the output intensity of a ray segment of length Δs is

$$L(s_i)\left(1 - e^{-\tau(s_i)\Delta s}\right) = L(s_i)(1 - \zeta_i) \tag{3.12}$$

Using Equation 3.12 for g_i results in a more accurate approximation of the integral. Nevertheless, we may precompute Equation 3.12 just as we can precompute ζ_i. Furthermore, consider what happens when we substitute Equation 3.12 for the g_is in Equation 3.11.

$$I(D) \approx L_n(1 - \zeta_n) + \zeta_n(L_{n-1}(1 - \zeta_{n-1}) + \cdots \zeta_2(L_1(1 - \zeta_1) + \zeta_1 I_0)\cdots) \tag{3.13}$$

We can use graphics hardware to perform the basic operation $L_i(1 - \zeta_i) + \zeta_i(\cdots)$ used in Equation 3.13 as described in Section 2.2.4.

Volume rendering systems use the Riemann sum method most often when sampling of the data set along viewing rays is convenient and does not result in a large loss of information. Such situations occur when rendering rectilinear grids using methods such as those described in Section 3.1.3.

3.2.2 Average Luminance and Attenuation

Max, Hanrahan, and Crawfis [64] and Shirley and Tuchman [86] independently developed a method for approximating the volume rendering integral for unstructured grids and other arenas where resampling is not practical.

The method starts with the (impractical) assumption that the luminance along the ray is constant. Plugging this constraint in the volume rendering integral we get

$$\begin{aligned} I(D) &= I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L\tau(s) e^{-\int_s^D \tau(t)dt} ds \\ &= I_0 e^{-\int_0^D \tau(t)dt} + L\left[e^{-\int_s^D \tau(t)dt}\right]_0^D \\ &= I_0 e^{-\int_0^D \tau(t)dt} + L\left(1 - e^{-\int_0^D \tau(t)dt}\right) \end{aligned} \tag{3.14}$$

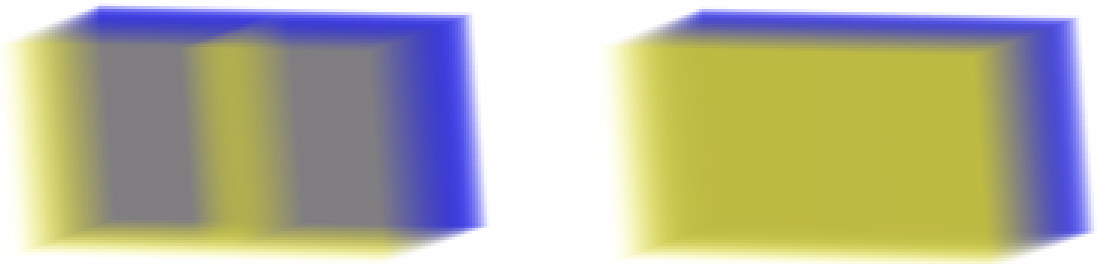
We can solve Equation 3.14 for any integrable function for $\tau(t)$. Recall from the discussion in Section 2.2.4 that we can perform piecewise integration on the viewing rays. Furthermore, it is convenient to break up the integral based on its intersection with cells. The assumption that the volume properties vary linearly through the cells is often (but not always) valid. Solving Equation 3.14 for a linear interpolation of the attenuation, we get

$$I(D) = I_0 e^{-\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} + L\left(1 - e^{-\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}}\right) \tag{3.15}$$

We still have a major problem with Equation 3.15: the luminance is constant. In general, we require the luminance to vary within cells just like the attenuation. Both [64] and [86] solve this problem by averaging the color over the length of the segment. When the color varies linearly like the luminance, this approximation yields

$$I(D) = I_0 e^{-\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} + \frac{L_\mathrm{b} + L_\mathrm{f}}{2}\left(1 - e^{-\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}}\right) \tag{3.16}$$

An interesting feature of Equation 3.16 is that it is equivalent to approximating the volume by averaging the luminance and attenuation and assuming the volume is



(a) Averaged color. (b) Linearly interpolated color.

Figure 3.12: An example in the error caused by averaging luminance in a viewing ray. Both images show two hexahedral cells. The image on the left averages the luminance in each ray. The image on the right has correct linear interpolation.

homogeneous. This can be verified by plugging in $L(s) = (L_\mathrm{b} + L_\mathrm{f})/2$ and $\tau(s) = (\tau_\mathrm{b}+\tau_\mathrm{f})/2$ into the volume integral for homogeneous volumes (given in Section 2.3.3).

The biggest cause of error is that caused by averaging the luminance. Figure 3.12 demonstrates the error that can occur. When we average the luminance, the color on the back faces of the cells bleeds in through the front. Furthermore, although the color should be constant along the front face of the volume, the approximation changes at the interface between the two cells. The change in colors causes **Mach bands** to be visible. Mach bands are lines introduced by the human visual system in places where color changes are discontinuous. Mach bands help the visual system detect the edges of objects.

3.2.3 Linear Interpolation of Luminance and Intensity

In Section 2.3.5, I solve the volume rendering integral for linearly varying luminance and attenuation. I parameterize the integral with the front and back values of the

luminance and attenuation, as shown in the following two equations.

$$L(s) = L_{\mathrm{b}}\frac{D-s}{D} + L_{\mathrm{f}}\frac{s}{D} \tag{3.17}$$

$$\tau(s) = \tau_{\mathrm{b}}\frac{D-s}{D} + \tau_{\mathrm{f}}\frac{s}{D} \tag{3.18}$$

The solution to the volume rendering integral using the linear terms demonstrated in Equations 3.17 and 3.18 is complicated. If we wish to compute the volume rendering integral using only finite, real values, we need at least three forms of the solution. The first form has real and finite terms when $\tau_{\mathrm{b}} > \tau_{\mathrm{f}}$.

$$\begin{aligned} I(D) = I_0 e^{-D\frac{\tau_{\mathrm{b}}+\tau_{\mathrm{f}}}{2}} + L_{\mathrm{f}} - L_{\mathrm{b}} e^{-D\frac{\tau_{\mathrm{b}}+\tau_{\mathrm{f}}}{2}} \\ +(L_{\mathrm{b}} - L_{\mathrm{f}})\frac{1}{\sqrt{D(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}} e^{\frac{D}{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}\tau_{\mathrm{f}}^2}\sqrt{\frac{\pi}{2}} \\ \left[\mathrm{erf}\left(\tau_{\mathrm{b}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}}\right) - \mathrm{erf}\left(\tau_{\mathrm{f}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}}-\tau_{\mathrm{f}})}}\right)\right] \end{aligned} \tag{3.19}$$

The second form has real and finite terms when $\tau_{\mathrm{b}} < \tau_{\mathrm{f}}$.

$$\begin{aligned} I(D) = I_0 e^{-D\frac{\tau_{\mathrm{f}}+\tau_{\mathrm{b}}}{2}} + L_{\mathrm{f}} - L_{\mathrm{b}} e^{-D\frac{\tau_{\mathrm{f}}+\tau_{\mathrm{b}}}{2}} \\ +(L_{\mathrm{b}} - L_{\mathrm{f}})\frac{1}{\sqrt{D(\tau_{\mathrm{f}}-\tau_{\mathrm{b}})}} e^{-\frac{D}{2(\tau_{\mathrm{f}}-\tau_{\mathrm{b}})}\tau_{\mathrm{f}}^2}\sqrt{\frac{\pi}{2}} \\ \left[\mathrm{erfi}\left(\tau_{\mathrm{f}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{f}}-\tau_{\mathrm{b}})}}\right) - \mathrm{erfi}\left(\tau_{\mathrm{b}}\frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{f}}-\tau_{\mathrm{b}})}}\right)\right] \end{aligned} \tag{3.20}$$

The third form is valid when $\tau_{\mathrm{b}} = \tau_{\mathrm{f}} = \tau$.

$$I(D) = I_0 e^{-\tau D} + L_{\mathrm{b}}\left(\frac{1}{\tau D} - \frac{1}{\tau D}e^{-\tau D} - e^{-\tau D}\right) + L_{\mathrm{f}}\left(1 + \frac{1}{\tau D}e^{-\tau D} - \frac{1}{\tau D}\right) \tag{3.21}$$

Computing Equations 3.19 through 3.21 is not straightforward. First is the problem of computing the functions erf and erfi with high numerical accuracy. Second is the problem where the value in the brackets can become quite small while the value it multiplies with becomes exceptionally large, which leads to numerical instability. Williams, Max, and Stein [105] solve these problems, and I review their solutions here.

We first consider Equation 3.19. In particular, consider the last term. For convenience, let us use the groupings $\delta_{\mathrm{b}} \equiv \tau_{\mathrm{b}} \frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}} - \tau_{\mathrm{f}})}}$ and $\delta_{\mathrm{f}} \equiv \tau_{\mathrm{f}} \frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{b}} - \tau_{\mathrm{f}})}}$. The third term of Equation 3.19 becomes

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{1}{\sqrt{D(\tau_{\mathrm{b}} - \tau_{\mathrm{f}})}} e^{\delta_{\mathrm{f}}^2} \sqrt{\frac{\pi}{2}} \left[\mathrm{erf}\left(\delta_{\mathrm{b}}\right) - \mathrm{erf}\left(\delta_{\mathrm{f}}\right)\right] \tag{3.22}$$

In particular, note that if δ_{b} and δ_{f} are moderately large, then the value inside the brackets is close to zero and the exponent outside the brackets is large. The disparity between the two factors leads to numerical errors.

To compute the erf functions, Williams uses an approximation given by Press and colleagues [75]. (Press actually gives his approximation for the **complementary error function**, defined as $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$, but the conversion is trivial.) Press demonstrates approximating erf as

$$\mathrm{erf}(x) \cong 1 - u(x) e^{-x^2 + p(u(x))} \tag{3.23}$$

where $u(x) = \frac{1}{1+0.5x}$ and $p(z)$ is a ninth degree Chebyshev polynomial selected to give an accurate fit to the tail of the error function. Specifically, $p(z)$ is

$$\begin{aligned} p(z) = {} & -1.26551223 + z * (1.00002368 + z * (0.37409196 \\ & + z * (0.09678418 + z * (-0.18628806 + z * (0.27886807 \\ & + z * (-1.13520398 + z * (1.48851587 + z * (-0.82215223 \\ & + z * 0.17087277)))))))); \end{aligned} \tag{3.24}$$

Substituting Equation 3.23 into Equation 3.22, we get more flexibility.

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{1}{\sqrt{D(\tau_{\mathrm{b}} - \tau_{\mathrm{f}})}} e^{\delta_{\mathrm{f}}^2} \sqrt{\frac{\pi}{2}} \left[1 - u(\delta_{\mathrm{b}}) e^{-\delta_{\mathrm{b}}^2 + p(u(\delta_{\mathrm{b}}))} - 1 + u(\delta_{\mathrm{f}}) e^{-\delta_{\mathrm{f}}^2 + p(u(\delta_{\mathrm{f}}))}\right]$$

Now we cancel the 1s and move the exponent inside the brackets.

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{1}{\sqrt{D(\tau_{\mathrm{b}} - \tau_{\mathrm{f}})}} \sqrt{\frac{\pi}{2}} \left[u(\delta_{\mathrm{f}}) e^{p(u(\delta_{\mathrm{f}}))} - u(\delta_{\mathrm{b}}) e^{\delta_{\mathrm{f}}^2 - \delta_{\mathrm{b}}^2 + p(u(\delta_{\mathrm{b}}))}\right] \tag{3.25}$$

Equation 3.25 does not have the disparate factors as before and therefore is generally far more accurate when evaluated.

We next consider Equation 3.20. I rewrite the last term using the groupings $\delta'_{\mathrm{b}} \equiv \tau_{\mathrm{b}} \frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{f}} - \tau_{\mathrm{b}})}}$ and $\delta'_{\mathrm{f}} \equiv \tau_{\mathrm{f}} \frac{\sqrt{D}}{\sqrt{2(\tau_{\mathrm{f}} - \tau_{\mathrm{b}})}}$.

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{1}{\sqrt{D(\tau_{\mathrm{f}} - \tau_{\mathrm{b}})}} e^{-\delta'^2_{\mathrm{f}}} \sqrt{\frac{\pi}{2}} \left[\mathrm{erfi}\left(\delta'_{\mathrm{f}}\right) - \mathrm{erfi}\left(\delta'_{\mathrm{b}}\right)\right] \tag{3.26}$$

Rather than compute erfi directly, Williams instead computes **Dawson's integral**, defined as $D(x) \equiv e^{x^2} \int_0^x e^{y^2} dy$. Dawson's integral relates to erfi as $D(x) = \frac{1}{2}\sqrt{\pi} e^{-x^2} \mathrm{erfi}(x)$. Rybicki [82] gives a good numerical method for computing Dawson's integral. Press and colleagues [76] also summarize the method.

Substituting $\mathrm{erfi}(x) = \frac{2}{\sqrt{\pi}} e^{x^2} D(x)$ into Equation 3.26, we again get more flexibility.

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{1}{\sqrt{D(\tau_{\mathrm{f}} - \tau_{\mathrm{b}})}} e^{-\delta'^2_{\mathrm{f}}} \sqrt{\frac{\pi}{2}} \left[\frac{2}{\sqrt{\pi}} e^{\delta'^2_{\mathrm{f}}} D(\delta'_{\mathrm{f}}) - \frac{2}{\sqrt{\pi}} e^{\delta'^2_{\mathrm{b}}} D(\delta'_{\mathrm{b}})\right]$$

$$(L_{\mathrm{b}} - L_{\mathrm{f}}) \frac{\sqrt{2}}{\sqrt{D(\tau_{\mathrm{f}} - \tau_{\mathrm{b}})}} \left[D(\delta'_{\mathrm{f}}) - e^{\delta'^2_{\mathrm{b}} - \delta'^2_{\mathrm{f}}} D(\delta'_{\mathrm{b}})\right] \tag{3.27}$$

Equation 3.21 contains no form of the error function and is numerically stable enough to compute directly with the exception of when the quantity τD approaches zero. However, it is straightforward to show that $\lim_{\tau D = 0} I(D) = I_0$. Therefore, this is just a simple special case.

So, in summary of Williams and colleagues' work [105], we can accurately compute the volume rendering integral with linearly interpolated attenuation and luminance

with the following equation.

$$I(D) \cong \begin{cases} I_0 e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} + L_\mathrm{f} - L_\mathrm{b} e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} \\ \qquad +(L_\mathrm{b} - L_\mathrm{f})\frac{1}{\sqrt{D(\tau_\mathrm{b}-\tau_\mathrm{f})}}\sqrt{\frac{\pi}{2}} & \tau_\mathrm{b} > \tau_\mathrm{f} \\ \qquad \left[u(\delta_\mathrm{f})e^{p(u(\delta_\mathrm{f}))} - u(\delta_\mathrm{b})e^{\delta_\mathrm{f}^2-\delta_\mathrm{b}^2+p(u(\delta_\mathrm{b}))}\right] \\ I_0 e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} + L_\mathrm{f} - L_\mathrm{b} e^{-D\frac{\tau_\mathrm{b}+\tau_\mathrm{f}}{2}} \\ \qquad +(L_\mathrm{b} - L_\mathrm{f})\frac{\sqrt{2}}{\sqrt{D(\tau_\mathrm{f}-\tau_\mathrm{b})}} & \tau_\mathrm{b} < \tau_\mathrm{f} \\ \qquad \left[D(\delta'_\mathrm{f}) - e^{\delta'^2_\mathrm{b}-\delta'^2_\mathrm{f}} D(\delta'_\mathrm{b})\right] \\ I_0 e^{-\tau D} + L_\mathrm{b}\left(\frac{1}{\tau D} - \frac{1}{\tau D}e^{-\tau D} - e^{-\tau D}\right) \\ \qquad +L_\mathrm{f}\left(1 + \frac{1}{\tau D}e^{-\tau D} - \frac{1}{\tau D}\right) & (\tau_\mathrm{b} = \tau_\mathrm{f} = \tau) \cap (\tau D > 0) \\ I_0 & (\tau_\mathrm{b} = \tau_\mathrm{f} = \tau) \cap (\tau D = 0) \end{cases} \tag{3.28}$$

where $\delta_\mathrm{b} \equiv \tau_\mathrm{b}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{b}-\tau_\mathrm{f})}}$, $\delta_\mathrm{f} \equiv \tau_\mathrm{f}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{b}-\tau_\mathrm{f})}}$, $\delta'_\mathrm{b} \equiv \tau_\mathrm{b}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}}$, $\delta'_\mathrm{f} \equiv \tau_\mathrm{f}\frac{\sqrt{D}}{\sqrt{2(\tau_\mathrm{f}-\tau_\mathrm{b})}}$, $u(x) = \frac{1}{1+0.5x}$, $p(x)$ is defined in Equation 3.24, and $D(x)$ is Dawson's integral.

The listing for a fragment program that calculates Equation 3.28 resides in Appendix B.3.

3.2.4 Gaussian Attenuation

When performing scientific volume visualization, most systems simply use 1D transfer functions. That is, the volume defines a field of scalars in space, and the volume rendering feeds these scalars into a 1D **transfer function** that returns the luminance and attenuation to use in the volume. The previously reviewed color calculation methods reflect the use of 1D transfer functions.

However, Kniss, Kindlmann, and Hansen [46, 47] show that using multidimensional transfer functions with vectors containing a scalar and its first and second derivatives is capable of providing useful renderings that cannot be performed with only 1D transfer functions. Furthermore, Kniss and colleagues [50] and Tzeng, Lum, and Ma [94] demonstrate that it is possible and useful to apply multidimensional transfer functions to vector data.

Kniss and colleagues [50] show how to build many useful multidimensional transfer functions using the **Gaussian** function (often known also as the **normal distribution**). Kniss defined the Gaussian transfer function of a vector of d dimensions as

$$\mathrm{GTF}_{\vec{\mu},\mathbf{K}}(\vec{v}) = \exp\left(-(\vec{v}-\vec{\mu})^{\mathrm{T}}\mathbf{K}^{\mathrm{T}}\mathbf{K}(\vec{v}-\vec{\mu})\right) \tag{3.29}$$

where $\vec{v}$ is the input vector, $\vec{\mu}$ is mean of the distribution, and $\mathbf{K}$ is a rotational matrix. A traditional Gaussian function has a scaling term of $1/\left((2\pi)^{d/2}|\mathbf{K}^{-1}\mathbf{K}^{-\mathrm{T}}|\right)$, which scales the function such that the area under the curve is 1. However, this property is not useful for transfer functions and Kniss therefore drops it.

Kniss interpolates the attenuation as

$$\tau(t) = \tau\mathrm{GTF}_{\vec{\mu},\mathbf{K}}\left(\vec{v}_1 + t(\vec{v}_2 - \vec{v}_1)\right) \tag{3.30}$$

Kniss then plugs Equation 3.30 into the volume rendering integral proposed by Max, Hanrahan, and Crawfis [64]. I also discuss this integral in Section 2.3.4. I give the equation for the integral, Equation 2.18, on page 26. Plugging Equation 3.30 into Equation 2.18, we get

$$I(D) = I_0 e^{-\tau D\int_0^1 \mathrm{GTF}_{\vec{\mu},\mathbf{K}}(\vec{v}_1+t(\vec{v}_2-\vec{v}_1))dt} + L\left(1 - e^{-\tau D\int_0^1 \mathrm{GTF}_{\vec{\mu},\mathbf{K}}(\vec{v}_1+t(\vec{v}_2-\vec{v}_1))dt}\right) \tag{3.31}$$

Kniss and colleagues [49] solve the integral

$$\int_0^1 \mathrm{GTF}_{\vec{\mu},\mathbf{K}}\left(\vec{v}_1 + t(\vec{v}_2 - \vec{v}_1)\right) dt \tag{3.32}$$

The solution they give is

$$\frac{\sqrt{\pi}}{2}\frac{S}{\|\vec{d}\|}\left(\operatorname{erf}(B)-\operatorname{erf}(A)\right) \tag{3.33}$$

where

$$A=\frac{\vec{d}\cdot\vec{v}_1'}{\|\vec{d}\|},\qquad B=\frac{\vec{d}\cdot\vec{v}_2'}{\|\vec{d}\|}=A+\|\vec{d}\|,\qquad S=e^{-\|\vec{v}_1'\|+A^2}$$

$$\vec{d}=\vec{v}_2'-\vec{v}_1',\qquad \vec{v}_1'=\mathbf{K}(\vec{v}_1-\vec{\mu}),\qquad \vec{v}_2'=\mathbf{K}(\vec{v}_2-\vec{\mu})$$

Once Equation 3.32 is computed (using Equation 3.33), computing the rest of Equation 3.31 is trivial. Of course, Equation 3.31 assumes a constant luminance. Kniss uses a weighted sum to simulate luminance interpolated on Gaussian curves; however, this approximation can lead to the same color bleeding demonstrated in Figure 3.12 on page 56.

3.2.5 Pre-Integration

Röttger, Kraus, and Ertl [81] use a clever technique called **pre-integration**. Their approach is to store the result of the volume rendering integral for all possible thickness, luminance, and attenuation parameters in a table for rapid lookup. Of course, doing this calculation for even the linear solution (Equation 3.28 on page 60) requires five independent variables. Creating a five dimensional table with high enough fidelity is impractical.

However, Röttger and colleagues' target applications are those pertaining to scientific visualization. As such, the volumetric parameters of the models they are rendering are specified by scalar information from a physical simulation, such as pressure or temperature, that are mapped to rendering parameters (luminance and attenuation) through a transfer function. Figure 3.13 demonstrates the process of converting scalar values to volume properties to color. By using their lookup table

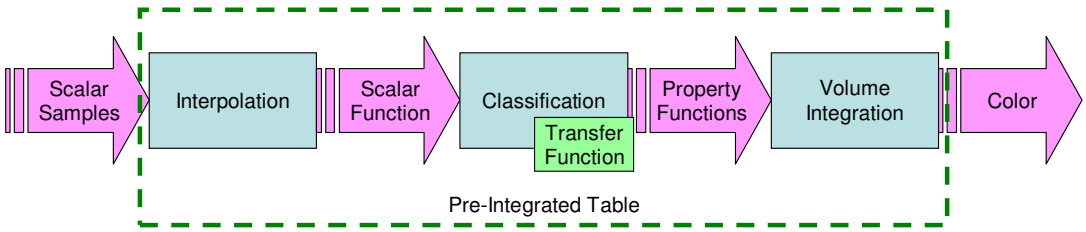


Figure 3.13: The calculations performed by a pre-integration table lookup. The pre-computed values in the table are performed over all the functional units encapsulated in the dashed green box.

for both **classification** (converting scalars to rendering properties) and computation of the volume rendering integral enables them to reduce the dimensionality of their lookup table to three. The dashed box in Figure 3.13 shows the scope of the calculation of the pre-integrated lookup. Röttger and colleagues provide also an approximation that requires only a two dimensional lookup table. (The approximation is improved by Guthe and colleagues [33] using a succession of 2D tables.)

The major advantage of the pre-integration technique is that it is fast (a texture lookup) and potentially can handle any interpolation of volumetric properties including an arbitrary number of isosurfaces. Engel, Kraus, and Ertl [22] argue (correctly) that interpolating the model's scalar values separately from the volumetric properties can reduce aliasing introduced by the interpolation. Ultimately, we can perform the pre-integration approach on commodity graphics hardware and can compute the volume rendering integral with appropriate interpolation in real time.

One disadvantage of the pre-integration approach is that the transfer function must be constant. If the transfer function ever changes, then an application has to recalculate the volume integral for every pair of entries in the table, often taking several minutes. This precludes the possibility of interactively changing the transfer function. Röttger and Ertl [80] improve the technique somewhat by performing the recalculation of the pre-integrated values on the graphics hardware, thereby reducing

the time from minutes to seconds.

Another potentially serious problem with pre-integration is that we can use only one-dimensional transfer functions without introducing impractically high table dimensions. However, Kniss, Kindlmann, and Hansen [46, 47] introduce effective tools that use 2- and 3-dimensional transfer functions for data exploration and presentation. Recent work has made advances in using transfer functions of even higher dimensions [50, 94].

Furthermore, the use of pre-integration precludes the use of any view-dependent rendering effects. These include several non-photorealistic rendering techniques [21, 43, 44] and global illumination [19, 37, 48, 51, 112, 113]. Therefore, the pre-integration technique is useful for many systems, but it has fundamental flaws that can hamper visualization tasks.

3.3 Summary

As I have shown in this chapter, the previous fifteen years have seen great strides in volume rendering. Although commodity graphics hardware still does not directly support volumetric primitives, several techniques utilize graphics hardware to perform some or all of the volume rendering. However, as the capabilities of graphics hardware continue to improve, we must continue to look for new ways to take advantage of them.

In addition, many approaches exist for computing the volume rendering integral. However, they all have flaws. Averaging luminance is fast but can be quite inaccurate. We can perform linear interpolation of luminance accurately, but it is far slower than other ray integration methods. Pre-integration computes the volume rendering integral outside of the rendering loop, so the system is capable of performing accurate

computations without affecting the rendering rate. However, the size of the lookup table limits the accuracy. Furthermore, we must rebuild the table every time the transfer function changes, which can be often. None of these algorithms can perform computations that are both fast and accurate for changes in viewing position and transfer function.

Chapter 4

Cell Projection

In this chapter, I discuss a fast, hardware-accelerated method for projecting tetrahedra. My algorithm runs efficiently on the current generation of graphics hardware, which is called the DirectX-9 class of graphics cards [61]. Furthermore, I expect the method will be applicable for many generations to come.

I start this chapter by reviewing the cell projection algorithm on which I base my work. I then make improvements on the algorithm that shall significantly improve the rendering speed. Finally, I make extensions to the algorithm that can increase the overall accuracy of the rendering system.

4.1 Quick Review of View Independent Cell Projection

I base my algorithm on the view independent cell projection of Weiler, Kraus, and Ertl [98, 100]. Like the ever-popular Projected Tetrahedra algorithm [86], view independent cell projection is a projection-based algorithm that takes advantage of

the optimized rasterizing of graphics hardware. However, view independent cell projection also takes advantage of programmable vertex hardware so that the CPU no longer must transform points or classify tetrahedron projections. I give an overview of this algorithm in Section 3.1.2 starting on page 39. I build on that review in this section by defining the procedures required.

I first give the procedure for projecting a single tetrahedron. Obviously, it is the application's responsibility to loop over all tetrahedra and render each one. A tetrahedron, by definition, has four triangular faces and four vertices. I assume that each vertex has a scalar associated with it. Furthermore, I assume that we interpolate the scalars linearly throughout the tetrahedron. Given these assumptions, the scalar gradient is constant. Section 3.1.2 discusses how to compute this gradient on page 43.

Models sometimes define scalar data on a per-cell basis. In this case, all four vertices of the tetrahedron have the same scalar value and the gradient is the zero vector. We can optimize the procedures I give for this case, but the optimization is trivial and I will not discuss it. It is possible also for the model to define the scalars with arbitrary parametric functions (in so-called nonlinear cells). The current best method for rendering such cells is to decompose the nonlinear cells into linear pieces [42].

The VICP-ProjectTet procedure formalizes the part of the view independent cell projection algorithm that runs on the CPU.

VICP-PROJECTTET(T)

```
1  g⃗ ← gradient of scalars in tetrahedron T
2  for each face F of tetrahedron T
3       do Send BEGIN_TRIANGLE to graphics card
4          for i ← 0 to 2
5               do v⃗ ← vertex i of face F
6                  s_v ← scalar value at v⃗
7                  P_v ← plane for face opposite v⃗
8                  Send (v⃗, s_v, P_v, g⃗, i) to graphics card
9          Send END_TRIANGLE to graphics card
```

The VICP-PROJECTTET procedure is quite simple. By design, the VICP-PROJECTTET procedure does little more than send data to the graphics card. The overhead of this part of the algorithm is not the amount of computation performed on the CPU, which is almost nothing, but rather the amount of data passed to the graphics card. All memory passing from the CPU to the GPU must pass through a bus, which often leads to a bottleneck. I therefore count how much memory VICP-PROJECTTET transfers to the graphics card.

Although I show VICP-PROJECTTET specifically sending begin and end triangle events in lines 3 and 9, respectively, we more commonly simply specify that every three vertices makes a triangle. Because the begin-triangle and end-triangle events are not specifically sent to the graphics hardware, I will not count them.

In line 8, VICP-PROJECTTET sends $(\vec{v}, s_v, P_v, \vec{g}, i)$ to the graphics card. s_v and i are scalars and thus require one component each. $\vec{v}$ and $\vec{g}$ are each 3-vectors. A plane equation such as P_v is often parameterized with four components, but Weiler, Kraus, and Ertl [98] demonstrate how to parameterize it with only three components. Therefore, assuming we represent all the components with floating point values, each call to line 8 passes 11 floats to the graphics hardware. Line 8 is called three times

per triangle, and there are four triangles per tetrahedron, so 132 floats are passed to the graphics hardware per tetrahedron in all.[1]

The interesting calculations in view independent cell projection begin in the vertex program. The vertex program, defined as VICP-VertexProg, takes the parameters for a single vertex as input (as sent from VICP-ProjectTet) and returns modified parameters that the rasterizer interpolates.

VICP-VertexProg($\vec{v}, s_v, P_v, \vec{g}, i$)

1 $\vec{v}' \leftarrow \vec{v}$ modified by current transform
2 $P_v' \leftarrow P_v$ modified by current transform
3 $\vec{d} \leftarrow [0, 0, 0]$
4 $\vec{s} \leftarrow [s_v, s_v, s_v]$
5 $R \leftarrow$ ray originating at $\vec{v}'$ and pointing in view direction
6 $(\vec{d_i}, \vec{s_i}) \leftarrow$ IntersectBackFace($R, P_v', s_v, \vec{g}$)
7 **return** $(\vec{v}', s_v, \vec{d}, \vec{s})$

VICP-VertexProg generates the vectors $\vec{d}$ and $\vec{s}$. Each entry in the $\vec{d}$ vector gives the distance, along the view vector, to one of the opposite faces (an opposite face being one of the three faces of the tetrahedron not being projected). Each entry in the $\vec{s}$ vector gives the scalar value on the corresponding opposite face. Each vertex touches three faces. One of the faces is the one being projected and is not included in the $\vec{d}$ and $\vec{s}$ vectors. The other two faces correspond to 0 entries in $\vec{d}$ and s_v entries in $\vec{s}$. Lines 3 and 4 initialize the two vectors.

In line 6, VICP-VertexProg computes the intersection of the viewing ray with the one face not touching the vertex. The procedure IntersectBackFace performs

[1] Weiler and colleagues [100] actually pass plane information for three faces instead of one face and the index. This makes 16 floats per vertex or 192 floats per tetrahedron. The added floating point values allow the fragment program to perform the intersection of each viewing ray with all three potential exiting faces.

this intersection. It returns the distance to the intersection and the scalar value at the intersection, both of which are placed in the appropriate index of the $\vec{d}$ and $\vec{s}$ vectors. I do not specifically give the implementation of IntersectBackFace, but Equations 3.3 and 3.4 in Section 3.1.2 on page 42 provide the mathematics for the operation.

The rasterizer interpolates the values of $\vec{d}$ and $\vec{s}$ across the front face. In other words, the rasterizer is interpolating the distance to and scalar value of viewing ray intersections to all potential back faces. It is the job of the fragment program to pick the values for the actual exit point.

VICP-FragmentProg$(s_v, \vec{d}, \vec{s})$
1 $i \leftarrow$ the index such that $\left(\vec{d}_i > 0\right) \cap \left(\forall j, \vec{d}_j > 0 \Rightarrow \vec{d}_j \geq \vec{d}_i\right)$
2 **return** IntegrateRay$(s_v, \vec{s}_i, \vec{d}_i)$

VICP-FragmentProg picks the correct exit point by examining the distance to each face. Specifically, the correct exit point has a distance that is the minimum of all those greater than zero. Once it determines the correct exit point, VICP-FragmentProg needs only perform ray integration. I discuss ray integration in Chapter 5.

4.2 GPU-CPU Balanced Cell Projection

In this section, I modify the view independent cell projection algorithm of Weiler, Kraus, and Ertl [98] by removing one constraint: view independence. In order for the projection to be view independent, the algorithm must perform all the calculation on the graphics card. Furthermore, the algorithm must pass all the data required for the calculation to the graphics card. Because I plan to use optical models that are view dependent, having a view independent cell projection is not helpful.

I modify Weiler, Kraus, and Ertl's algorithm by moving some of the calculations from the vertex program back into the CPU. The changes I make have minimal effect on either the CPU or GPU running time. Rather, my changes minimize the amount of memory that we must transfer from the CPU to the GPU.

BALANCED-PROJECTTET(T)

1 $\vec{g} \leftarrow$ gradient of scalars in tetrahedron T
2 **for** $i \leftarrow 0$ **to** 3
3 **do** $\vec{v} \leftarrow$ vertex i of tetrahedron T
4 $s_{\mathrm{f}} \leftarrow$ scalar value at $\vec{v}$
5 $P_v \leftarrow$ plane for face opposite $\vec{v}$
6 $R \leftarrow$ ray originating at $\vec{v}$ and pointing in view direction
7 $(d, s_{\mathrm{b}}) \leftarrow$ INTERSECTBACKFACE$(R, P_v, s_{\mathrm{f}}, \vec{g})$
8 Send $(\vec{v}, d, s_{\mathrm{f}}, s_{\mathrm{b}}, i)$ to graphics card at index i
9 Send RENDERTRIANGLESTRIP + 6 indices

Let us compare the differences between VICP-PROJECTTET and BALANCED-PROJECTTET. First, the balanced cell projection calls INTERSECTBACKFACE here on the CPU rather than in the vertex program. Thus, rather than send the vectors for P_v and $\vec{g}$, we send the two scalars d and s_{b}. Second, the balanced cell projection sends vertex information 4 times rather than 12. To do this, I use an indexing mode that allows me to reuse vertex information among faces. Vertex arrays and the vertex buffer object extension support indexing mode. The reason we need to send the vertex information twelve times in VICP-PROJECTTET and only four times in BALANCED-PROJECTTET becomes clear when we analyze BALANCED-VERTPROG.

BALANCED-VERTPROG$(\vec{v}, d, s_{\mathrm{f}}, s_{\mathrm{b}}, i)$

1 $\vec{v}' \leftarrow \vec{v}$ modified by current transform
2 $\vec{d} \leftarrow [0, 0, 0, 0]$
3 $\vec{d}_i \leftarrow d$
4 $\vec{s} \leftarrow [s_{\mathrm{f}}, s_{\mathrm{f}}, s_{\mathrm{f}}, s_{\mathrm{f}}]$
5 $\vec{s}_i \leftarrow s_{\mathrm{b}}$
6 **return** $(\vec{v}', s_{\mathrm{f}}, \vec{d}, \vec{s})$

BALANCED-VERTPROG differs from VICP-VERTPROG in two ways. First, the BALANCED-VERTPROG procedure does not compute INTERSECTBACKFACE because BALANCED-PROJECTTET already computed it. Second, $\vec{d}$ and $\vec{s}$ are 4-vectors rather than 3-vectors. One of the values in the 4-vector corresponds to the face that the rasterizer interpolates. Thus, we know the corresponding values in $\vec{d}$ and $\vec{s}$ will be 0 and s_{f}, respectively. However, commodity graphics hardware has redundant arithmetic units to handle 4-vectors, so the extra computation costs us nothing.

The 4-vectors $\vec{d}$ and $\vec{s}$ maintain the intersection of the viewing ray with all four faces of the tetrahedron. The four faces do not change with the face being projected, and so the indexing of the faces may remain consistent. When dealing with 3-vectors as in VICP-VERTPROG, the indexing of these vectors must change with each face being projected. This is why VICP-PROJECTTET has to send data for twelve vertices whereas BALANCED-PROJECTTET has to send data for only four vertices. We thus reduce the amount of data sent to the GPU from 132 floats to 28 floats. BALANCED-PROJECTTET has also to send six indices, but these indices can remain constant while a sorting algorithm reorders the vertex information. Therefore, we can store them directly on the graphics card with vertex buffer objects, and we do not have to send them every frame.

BALANCED-FRAGMENTPROG$(s_v, \vec{d}, \vec{s})$
1 $i \leftarrow$ the index such that $\left(\vec{d}_i > 0\right) \cap \left(\forall j, \vec{d}_j > 0 \Rightarrow \vec{d}_j \geq \vec{d}_i\right)$
2 **return** INTEGRATERAY$(s_v, \vec{s}_i, \vec{d}_i)$

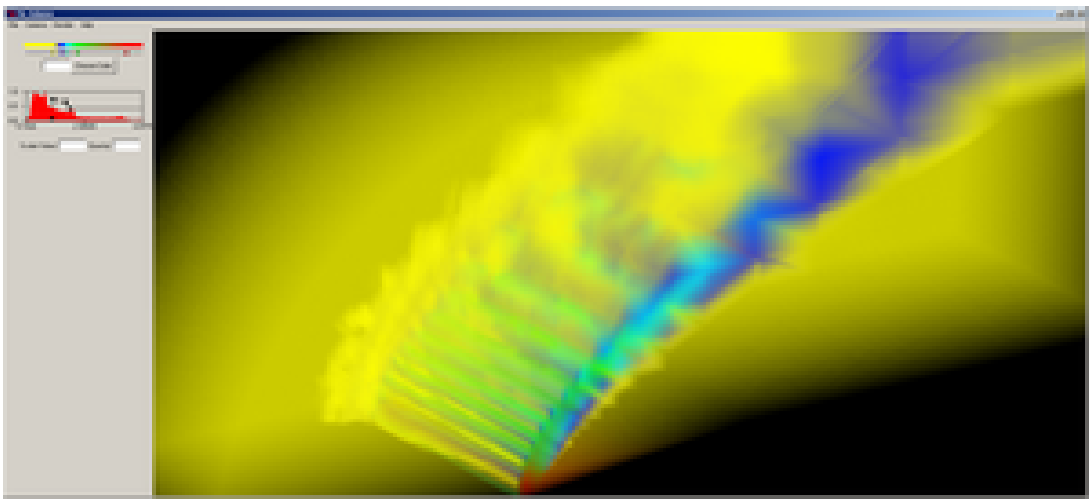
BALANCED-FRAGMENTPROG looks exactly like VICP-FRAGMENTPROG. The only difference is that $\vec{d}$ and $\vec{s}$ are 4-vectors instead of 3-vectors. Again, the extra entry holds the intersection for the face being projected. That entry in $\vec{d}$ will be zero, so BALANCED-FRAGMENTPROG will never select it.

4.3 Adaptive Transfer Function Sampling

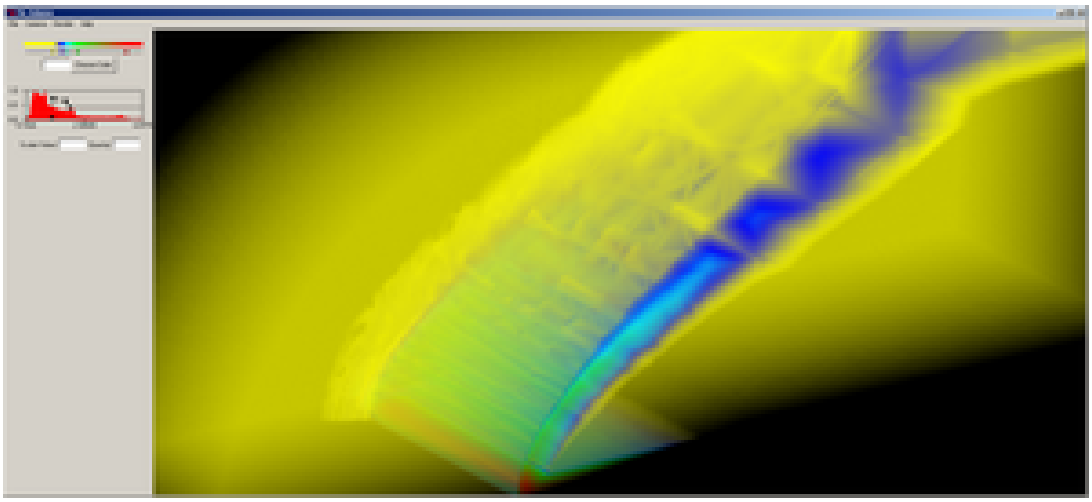
One major problem of volume rendering systems is that of aliasing of the transfer function. Typically, we sample the transfer function at the vertices of cells and interpolate the colors. However, this sampling often completely misses transitions in the transfer function. As an example, consider Figure 4.1(a). Because it samples the transfer function at only cell vertices, the renderer completely misses sharp transitions in the transfer function within cells, leaving a blocky, blurry mess. Compare this result to the appropriate transfer function sampling in Figure 4.1(b).

Engel, Kraus, and Ertl [22] solve this problem using pre-integration. By performing the integration offline, the rendering system can afford to sample the transfer function tightly independent of how the model samples scalars. However, I choose to avoid pre-integration because of the high computational overhead required every time the transfer function changes. Furthermore, the accuracy of pre-integration is low and pre-integration precludes the use of multidimensional transfer functions [47, 50], non-photorealistic rendering effects [21, 43, 44], and global illumination [19, 37, 51, 113].

Williams, Max, and Stein [105] solve this same problem by splitting cells. They



(a) Linearly interpolated colors and opacities.



(b) Adaptively sampled transfer function.

Figure 4.1: The effect of aliasing of the transfer function. Both images have the same transfer function, which has a sharp opacity transition to highlight an isosurface. The rendering on the top samples the transfer function at only the vertices of the cells, which induces aliasing. The rendering on the bottom adaptively samples the transfer function.

define their transfer functions as piecewise linear functions. Each **control point**, a point where the transfer function is nonlinear, defines an isosurface. They split the cells on these isosurfaces, yielding a linear interpolation of rendering parameters within the split cells. The problem with the Williams, Max, and Stein approach is that it also introduces a high overhead when the transfer function changes.

My approach is similar to that of Williams, Max, and Stein in that I split cells on the isosurfaces of transfer-function control points. Except, instead of splitting a cell geometrically, I clip the cell on the graphics card. To do the clipping I need to pass two more pieces of information, the two control points that are clipping the tetrahedron, to the graphics card. (Note that this added information increases the data sent to the card to 36 floats per tetrahedron.) If any control points lay within the scalar range of a tetrahedron, I render the tetrahedron multiple times with varying clipping parameters until the entire cell is rendered. The ATFS-ProjectClippedTet procedure formalizes how, given a tetrahedron and two control points, we can send the data to the graphics card.

ATFS-ProjectClippedTet$(T, c_\mathrm{f}, c_\mathrm{b})$

1 $\vec{g} \leftarrow$ gradient of scalars in tetrahedron T
2 **for** $i \leftarrow 0$ **to** 3
3 **do** $\vec{v} \leftarrow$ vertex i of tetrahedron T
4 $s_\mathrm{f} \leftarrow$ scalar value at $\vec{v}$
5 $P_v \leftarrow$ plane for face opposite $\vec{v}$
6 $R \leftarrow$ ray originating at $\vec{v}$ and pointing in $\vec{view}$
7 $(d, s_\mathrm{b}) \leftarrow$ IntersectBackFace$(R, P_v, s_\mathrm{f}, \vec{g})$
8 $norm\text{-}s_\mathrm{f} \leftarrow (s_\mathrm{f} - c_\mathrm{f})/(c_\mathrm{b} - c_\mathrm{f})$
9 $norm\text{-}s_\mathrm{b} \leftarrow (s_\mathrm{b} - c_\mathrm{f})/(c_\mathrm{b} - c_\mathrm{f})$
10 Send $(\vec{v}, d, norm\text{-}s_\mathrm{f}, norm\text{-}s_\mathrm{b}, c_\mathrm{f}, c_\mathrm{b}, i)$ to graphics card at index i
11 Send RenderTriangleStrip + 6 indices

The ATFS-ProjectClippedTet procedure is identical to the Balanced-ProjectTet procedure with two exceptions. First, the two scalar values of the transfer function control points (c_f and c_b) are passed to the graphics card. Second, rather than send the scalar values themselves, lines 8 and 9 normalize the scalars to the two control points.

Using normalized scalar values provides two features. One feature is that all parts of the tetrahedron that we do not clip will have a normalized scalar value in the range $[0, 1]$, making it easier to identify the clipped regions. Another feature is that because the luminance and attenuation parameters will vary linearly with the scalars in the range between the two control points, we can use the normalized scalar to interpolate these parameters. Thus, we can store the transfer function in the graphics card as a set of control points and then pass c_f and c_b indices to the appropriate control point. This mode removes any error that might occur with sampling the transfer function because it does not sample the transfer function.

Of course, we still need to determine the front and back transfer-function control points (c_f and c_b) before calling ATFS-ProjectClippedTet. The procedure ATFS-ProjectTet extracts the appropriate control points and renders the clipped tetrahedra in back to front order.

ATFS-ProjectTet(T, *transfer-func*)

```
1  c_min ← largest control point in transfer-func that is ≤ smallest scalar in T
2  c_max ← smallest control point in transfer-func that is ≥ largest scalar in T
3  g⃗ ← gradient of scalars in tetrahedron T
4  zgradient ← g⃗ · view⃗
5  if zgradient < 0
6     then c_b ← c_min
7          while c_b < c_max
8               do c_f ← smallest control point in transfer-func that is > c_b
9                  ATFS-ProjectClippedTet(T, c_f, c_b)
10                 c_b ← c_f
11    else c_b ← c_max
12         while c_b > c_min
13              do c_f ← largest control point in transfer-func that is < c_b
14                 ATFS-ProjectClippedTet(T, c_f, c_b)
15                 c_b ← c_f
```

ATFS-ProjectTet first determines the range of scalar values within tetrahedron T and over which transfer-function control points the range lies (lines 1 and 2). The procedure then determines in which direction to traverse the control points by taking a dot product of the scalar gradient with the view vector (lines 3 and 4). Both branches of the **if** statement starting on line 5 iterate over the control points and render each clipped piece of the tetrahedron in back-to-front order. If the blending requires front-to-back rendering rather than back-to-front rendering, then we can reverse the conditions of the **if** statement.

ATFS-VertProg($\vec{v}, d, \textit{norm-s}_\mathrm{f}, \textit{norm-s}_\mathrm{b}, c_\mathrm{f}, c_\mathrm{b}, i$)

1 $\vec{v}' \leftarrow \vec{v}$ modified by current transform
2 $\vec{d} \leftarrow [0, 0, 0, 0]$
3 $\vec{d}_i \leftarrow d$
4 $\vec{norm\text{-}s} \leftarrow [\textit{norm-s}_\mathrm{f}, \textit{norm-s}_\mathrm{f}, \textit{norm-s}_\mathrm{f}, \textit{norm-s}_\mathrm{f}]$
5 $\vec{norm\text{-}s}_i \leftarrow \textit{norm-s}_\mathrm{b}$
6 $\textit{iso-dist} \leftarrow d/(\textit{norm-s}_\mathrm{f} - \textit{norm-s}_\mathrm{b})$
7 **return** $(\vec{v}', \vec{d}, \textit{norm-s}_\mathrm{f}, \vec{norm\text{-}s}, c_\mathrm{f}, c_\mathrm{b}, \textit{iso-dist})$

The vertex program for adaptive transfer function sampling, ATFS-VertProg, is much like Balanced-VertProg. ATFS-VertProg, of course, passes also the two control points. In addition, it calculates the distance between the two isosurfaces at the control points in line 6. These two isosurfaces are parallel planes, and thus does not change throughout the tetrahedron. However, by performing the calculation in the vertex program, we can avoid transmitting more information from the CPU to the GPU.

The way ATFS-VertProg determines the distance is not straightforward. It is constrained to use the information already passed to it. From lines 8 and 9 in ATFS-ProjectClippedTet, we know that $\textit{norm-s} = (s - c_\mathrm{f})/(c_\mathrm{f} - c_\mathrm{b})$. Consider the difference of the two normalized scalars.

$$\begin{aligned} \textit{norm-s}_\mathrm{f} - \textit{norm-s}_\mathrm{b} &= \frac{s_\mathrm{f} - c_\mathrm{f}}{c_\mathrm{f} - c_\mathrm{b}} - \frac{s_\mathrm{b} - c_\mathrm{f}}{c_\mathrm{f} - c_\mathrm{b}} \\ &= \frac{s_\mathrm{f} - s_\mathrm{b}}{c_\mathrm{f} - c_\mathrm{b}} \end{aligned} \tag{4.1}$$

We know also that, given a linear interpolation of scalar values through space, the distance between two points in space along a viewing vector is proportional to the difference between the scalar values at those points. Therefore,

$$\frac{\textit{iso-dist}}{d} = \frac{c_\mathrm{f} - c_\mathrm{b}}{s_\mathrm{f} - s_\mathrm{b}}$$

where d, just like in ATFS-VertProg, is the distance between two points with scalar values s_{f} and s_{b}. It follows that

$$\textit{iso-dist} = d\frac{c_{\mathrm{f}} - c_{\mathrm{b}}}{s_{\mathrm{f}} - s_{\mathrm{b}}} \tag{4.2}$$

Combining Equation 4.1 and Equation 4.2, we get

$$\textit{iso-dist} = \frac{d}{\textit{norm-}s_{\mathrm{f}} - \textit{norm-}s_{\mathrm{b}}} \tag{4.3}$$

which line 6 of ATFS-VertProg clearly computes.

The isosurfaces of the control points are planes within each linearly interpolated tetrahedron. Ideally, we would like to use the clipping hardware of the graphics card to modify the geometry. Unfortunately, the clipping hardware can clip only polygons and the vertex processor cannot generate the extra vertices necessary to clip tetrahedra. Instead, we perform the clipping on each fragment. Consequently, ATFS-FragmentProg is more complicated than the previously defined fragment programs.

ATFS-FragmentProg($\vec{d}$, *norm-s*$_{\mathrm{f}}$, *norm-s*$_{\mathrm{b}}$, c_{f}, c_{b}, *iso-dist*)

```
1  i ← the index such that (d_i > 0) ∩ (∀j, d_j > 0 ⇒ d_j ≥ d_i)
2  norm-s_b ← norm-s_i
3  d ← d_i
4  param-c_f ← volume parameters for scalar c_f
5  param-c_b ← volume parameters for scalar c_b
6  if (norm-s_f > 1) ∪ (norm-s_b < 0)
7     then discard fragment
8  if norm-s_f < 0
9     then d ← d − (−norm-s_f) iso-dist
10         param-s_f ← param-c_f
11    else param-s_f ← LinearInterpolate(param-c_f, param-c_b, norm-s_f)
12 if norm-s_b > 1
13    then d ← d − (norm-s_b − 1) iso-dist
14         param-s_b ← param-c_b
15    else param-s_b ← LinearInterpolate(param-c_f, param-c_b, norm-s_b)
16 return IntegrateRay(param-s_f, param-s_b, d)
```

ATFS-FragmentProg starts the same as the previous fragment programs by determining the face that the viewing ray exits the tetrahedron through (line 1). It then retrieves normalized scalar at the exit point and the distance between ray entry and exit (lines 2 and 3). It follows by retrieving the volume parameters for the two transfer-function control points.[2] Lines 6 through 15 clip the ray segment.

To facilitate our discussion of per fragment tetrahedra clipping, consider Figure 4.2, which shows examples of clipped viewing-ray segments. Some rays, such as ray c, are not clipped at all. Other rays, such as rays a and d, will be completely re-

[2]Previous examples of fragment programs did not explicitly convert scalars to volume parameters. They instead delegated this process to IntegrateRay.

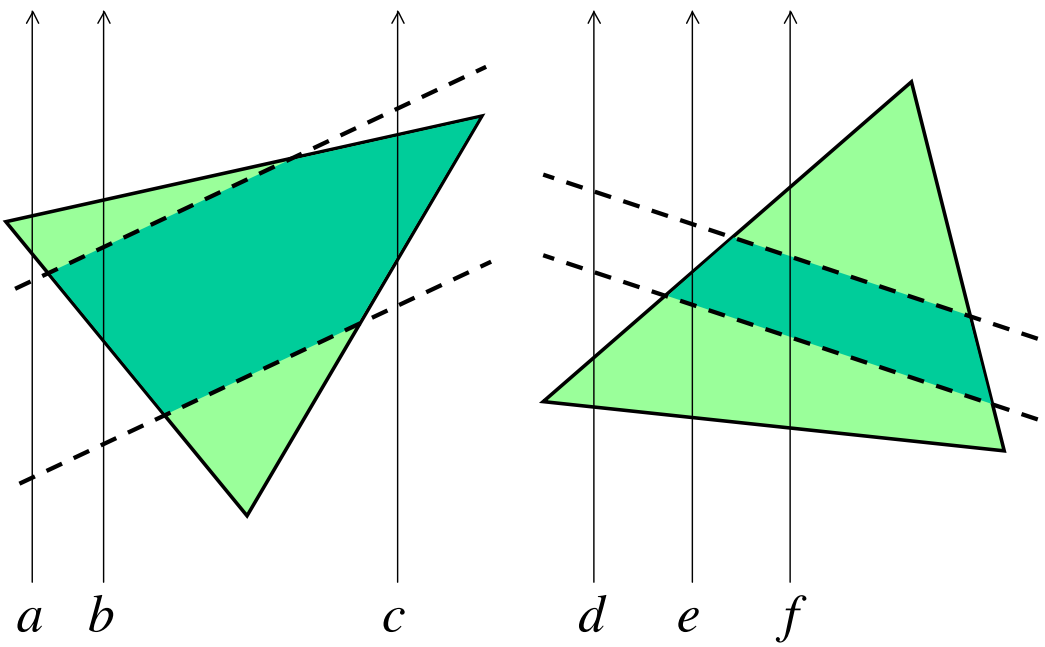


Figure 4.2: Tetrahedron clipping (reduced to a 2D example). Here I show two example triangles that I clip on a per fragment basis between the two isosurfaces. Six example viewing rays (labeled a–f) are given.

moved. Still others will be partially clipped such that they enter the front isosurface or exit the back isosurface or both, such as rays e, b, and f, respectively. As I explain the remainder of ATFS-FragmentProg, I will reference the rays in Figure 4.2.

The conditional on line 6 determines whether to clip the ray-tetrahedron intersection entirely. This type of clipping occurs if and only if the ray enters the tetrahedron behind the back isosurface (ray a) or exits the tetrahedron in front of the front isosurface (ray d). Although we could perform this test by comparing the position of intersections along the viewing ray, we can perform the same test by comparing the scalar values at these intersections, which are proportional. Because ATFS-ProjectClippedTet normalized the scalar values at the tetrahedron intersections, this test, like all the other tests, reduces to simply checking whether the normalized scalars are in the range $[0, 1]$. If the test determines to clip the entire ray,

ATFS-FragmentProg discards the fragment. That is, it writes nothing into the frame buffer.

The conditional on line 8 determines whether the front isosurface lies within the tetrahedron. This is true if and only if the ray enters the tetrahedron before it intersects the isosurface (and the first test fails). If the test determines to clip the front part of the ray (as in rays e and f), then ATFS-FragmentProg subtracts the distance between the ray entry and the isosurface from the segment length. The volume parameters at the front of the ray are also set to the value at the front isosurface. If the test determines to not clip the front part of the ray (as in rays b and c), then the volume parameters of the two isosurfaces are interpolated to get the scalar value at the surface of the tetrahedron. The conditional on line 12 performs the equivalent operations based on whether the back isosurface is inside the tetrahedron along the ray.

Appendices B.1 and B.2 list implementations of the vertex and fragment programs, respectively, required for tetrahedra projection with adaptive transfer function sampling.

4.4 Synopsis

Thanks to the assistance of efficient graphics hardware, the Projected Tetrahedra algorithm [86] was the fastest known unstructured-mesh rendering algorithm for over a decade. The view independent cell projection algorithm, developed by Weiler and colleagues [98, 100] and reviewed in Section 4.1, improves on the Projected Tetrahedra algorithm by taking advantage of graphics hardware capabilities that were not available when Projected Tetrahedra was developed.

In Section 4.2, I proposed changes to the view independent cell projection algo-

rithm that reduce the demands of the bandwidth between CPU and GPU. We shall see in Chapter 6 that these changes greatly improve the speed of the algorithm.

In Section 4.3, I added adaptive transfer function sampling to the algorithm. It divides the tetrahedra such that material properties vary linearly within each piece. This linear variance makes our integration of volume properties (discussed in the next chapter) far more accurate. This tetrahedron division ultimately slows down the algorithm. The changes send more data to the card and add tetrahedra to the rendering. However, we require better transfer function sampling such as this to achieve high quality renderings. We shall see the overall effect of adding adaptive transfer function sampling in Chapter 6.

Chapter 5

Ray Integration

In the previous chapter, I discussed how to take a collection of unstructured volume elements and sample them with a grid of viewing rays. In this chapter, I discuss how to take a collection of samples along a viewing ray and convert the material properties of the volume into the intensity of light that emits from the volume. I do not define a model for the volume in this chapter. I use the model defined in Chapter 2.

When computing the light intensity, I will not consider an infinite viewing ray, but rather I will consider a finite segment of a viewing ray. I assume that the volume properties vary linearly throughout the segment. The adaptive transfer function sampling method introduced in Section 4.3 ensures that the volume properties will vary linearly. Furthermore, I parameterize the volume properties as I do in Chapter 4: with the volume parameters at the front and back of the segment and the length of the segment.

I reviewed several competitive ray integration methods in Section 3.2. However, all of the methods either introduce artifacts (requiring excessive sampling of the volume) or are too computationally intensive to use in real time or interactive envi-

ronments. The ray integration methods I provide in this chapter have a balance of accuracy and computational complexity.

5.1 Linear Interpolation of Luminance

Of all the ray integration methods reviewed in Section 3.2, only the work of Williams and colleagues [105] (reviewed in Subsection 3.2.3) linearly interpolates the luminance. However, their solution is computationally intensive (see Appendix B.3 for a sample implementation). In this chapter, I present ray integration methods that also linearly interpolate the luminance, but can do so with far less computation. I do this by grouping terms in the volume rendering integral.

Let us start with the general volume rendering integral, Equation 2.2.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt}ds \tag{5.1}$$

We plug in a linear form for $L(s)$, $L(s) = L_\mathrm{b}(1 - \frac{s}{D}) + L_\mathrm{f}\frac{s}{D}$, and then group terms containing the parameters for luminance (L_b and L_f) obtaining

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D \left(L_\mathrm{b}\left(1 - \frac{s}{D}\right) + L_\mathrm{f}\frac{s}{D}\right)\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\int_0^D \left(1 - \frac{s}{D}\right)\tau(s)e^{-\int_s^D \tau(t)dt}ds + L_\mathrm{f}\int_0^D \frac{s}{D}\tau(s)e^{-\int_s^D \tau(t)dt}ds$$

We can further resolve the integrals through integration by parts.

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\left(\left(1 - \frac{s}{D}\right)e^{-\int_s^D \tau(t)dt}\Big|_0^D - \int_0^D -\frac{1}{D}e^{-\int_s^D \tau(t)dt}ds\right) + L_\mathrm{f}\left(\frac{s}{D}e^{-\int_s^D \tau(t)dt}\Big|_0^D - \int_0^D \frac{1}{D}e^{-\int_s^D \tau(t)dt}ds\right)$$

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + L_\mathrm{b}\left(-e^{-\int_0^D \tau(t)dt} + \frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt} ds\right) + L_\mathrm{f}\left(1 - \frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt} ds\right) \quad (5.2)$$

There is a significant amount of repetition of terms in Equation 5.2. We define the following two terms, each of which appear twice.

$$\zeta_{D,\tau(t)} \equiv e^{-\int_0^D \tau(t)dt} \quad (5.3)$$

$$\Psi_{D,\tau(t)} \equiv \frac{1}{D}\int_0^D e^{-\int_s^D \tau(t)dt} ds \quad (5.4)$$

Substituting Equations 5.3 and 5.4 into Equation 5.2 results in

$$I(D) = I_0 \zeta_{D,\tau(t)} + L_\mathrm{b}\left(\Psi_{D,\tau(t)} - \zeta_{D,\tau(t)}\right) + L_\mathrm{f}\left(1 - \Psi_{D,\tau(t)}\right) \quad (5.5)$$

Given $\zeta_{D,\tau(t)}$ and $\Psi_{D,\tau(t)}$, Equation 5.5 is a simple enough form to be computed in real time on a graphics processor. The following sections discuss the computation of $\zeta_{D,\tau(t)}$ and $\Psi_{D,\tau(t)}$.

5.2 Linear Interpolation of Attenuation

Consider the case when we interpolate the attenuation linearly, as is done in much of the volume rendering literature [63, 64, 86, 92, 102, 104, 105]. That is, $\tau(t) = \tau_\mathrm{b}(1-t) + \tau_\mathrm{f} t$. This section discusses solutions for ζ and Ψ and provides means of computing them when the attenuation is linear.

5.2.1 Computing ζ

Solving for $\zeta_{D,\tau(t)}$ is straightforward. Plugging in the linear form of $\tau(t)$ into Equation 5.3, we get the following.

$$\begin{aligned}\zeta_{D,\tau_\mathrm{b},\tau_\mathrm{f}} &= e^{-\int_0^D \left(\tau_\mathrm{b}\left(1-\frac{t}{D}\right)+\tau_\mathrm{f}\frac{t}{D}\right)dt} \\ &= e^{-\left(\tau_\mathrm{b}\left(t-\frac{t^2}{2D}\right)+\tau_\mathrm{f}\frac{t^2}{2D}\right)\Big|_0^D} \\ &= e^{-\left(\tau_\mathrm{b}\left(D-\frac{D}{2}\right)+\tau_\mathrm{f}\frac{D}{2}\right)} \\ &= e^{-\frac{D}{2}(\tau_\mathrm{b}+\tau_\mathrm{f})}\end{aligned} \tag{5.6}$$

Because Equation 5.6 resolves to such a simple expression, we can compute it directly in programmable fragment units (of DirectX 9 class graphics hardware [61]) with few instructions.

5.2.2 Computing Ψ

In contrast, using a linear form for $\tau(s)$ does not resolve $\Psi_{D,\tau(s)}$ to a simple, easily computed form.

$$\Psi_{D,\tau_\mathrm{b},\tau_\mathrm{f}} = \frac{1}{D}\int_0^D e^{-\int_s^D \left(\tau_\mathrm{b}\left(1-\frac{t}{D}\right)+\tau_\mathrm{f}\frac{t}{D}\right)dt} ds \tag{5.7}$$

However, the linear form of Ψ relies on only three variables: D, τ_b, and τ_f. It is therefore possible to precompute Ψ for all applicable $(D, \tau_\mathrm{b}, \tau_\mathrm{f})$ triples. Also, note that the Ψ table is ubiquitous. Once computed, its results are valid for any volume rendering application. Thus, we have the luxury of using numerical methods that may take hours or days.

Although loading a three-dimensional lookup table into a three-dimensional texture is possible, a two-dimensional table and a two-dimensional texture are preferable. Röttger and colleagues point out that the significantly lower memory requirements

of two-dimensional tables allow for much higher fidelity in the data they store [81] and that data retrieval from two-dimensional textures is often faster than that from three-dimensional textures on current graphics hardware [80]. Furthermore, not all current graphics hardware supports three-dimensional textures with floating point precision.

Fortunately, pre-integrated values of the Ψ function for linear attenuation coefficients may be stored in a two-dimensional table with *no approximations* apart from those introduced by sampling the function. Consider what happens when we change the limits of the integrals in Equation 5.7 to range between 0 and 1.

$$\begin{aligned}\Psi_{D,\tau_\mathrm{b},\tau_\mathrm{f}} &= \frac{1}{D}\int_0^D e^{-\int_s^D \left(\tau_\mathrm{b}\left(1-\frac{t}{D}\right)+\tau_\mathrm{f}\frac{t}{D}\right)dt} ds \\ &= \frac{1}{D}\int_0^D e^{-D\int_{s/D}^1 (\tau_\mathrm{b}(1-t)+\tau_\mathrm{f}t)dt} ds \\ &= \int_0^1 e^{-D\int_s^1 (\tau_\mathrm{b}(1-t)+\tau_\mathrm{f}t)dt} ds\end{aligned}$$

Next, we distribute D within the inner integral.

$$\Psi_{\tau_\mathrm{b}D,\tau_\mathrm{f}D} = \int_0^1 e^{-\int_s^1 (\tau_\mathrm{b}D(1-t)+\tau_\mathrm{f}Dt)dt} ds \tag{5.8}$$

Equation 5.8 demonstrates that we may store Ψ in a two-dimensional table by pre-computing Ψ for all applicable $(\tau_\mathrm{b}D, \tau_\mathrm{f}D)$ pairs. Before a lookup into this table may occur, we must compute the products $\tau_\mathrm{b}D$ and $\tau_\mathrm{f}D$. We can perform both multiplications in a single GPU vector operation, and we can reuse the products to compute ζ if we rewrite Equation 5.6 as $e^{-\frac{1}{2}(\tau_\mathrm{b}D+\tau_\mathrm{f}D)}$, so the multiplications are essentially free.

Once we compute ζ and determine Ψ via a lookup table, we can calculate Equation 5.5 directly to perform the ray integration. Because this method uses a table that holds the pre-integration of part of the volume rendering integral, I dub this method **partial pre-integration**.

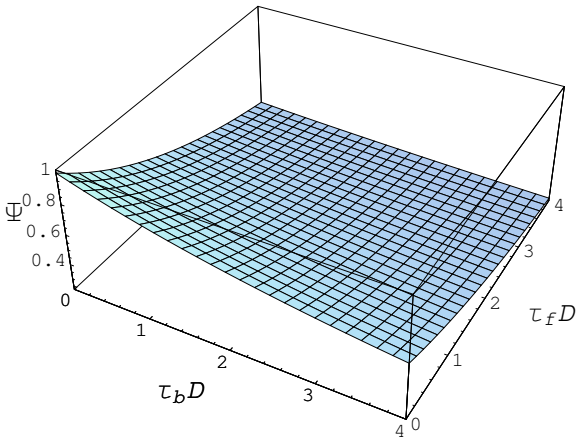


Figure 5.1: Plot of Ψ (Equation 5.8) against $\tau_{\mathrm{b}}D$ and $\tau_{\mathrm{f}}D$ for values in the range $[0, 4]$.

5.2.3 Domain of Ψ

In the previous section, I show that it is most prudent to build a two dimensional table with entries corresponding to $(\tau_{\mathrm{b}}D, \tau_{\mathrm{f}}D)$ pairs (Equation 5.8). However, valid values for τD lay in the range $[0, \infty)$. We must choose a subset of this range that provides good resolution for the most frequently encountered values of τD.

Figure 5.1 shows $\Psi_{\tau_{\mathrm{b}}D,\tau_{\mathrm{f}}D}$ plotted for parameters in the range $[0, 4]$. Despite the number of terms generated when solving Equation 5.8, we see that our plot is quite smooth. The absence of high frequencies makes storing the function over a finite domain in a table practical.

However, the partial pre-integration tables I have described thus far are not complete. For example, the domain for $\Psi_{\tau_{\mathrm{b}}D,\tau_{\mathrm{f}}D}$ shown in Figure 5.1 is not sufficient. Notice that $\Psi_{4,0} = 0.6$, whereas $\lim_{\tau_{\mathrm{b}}D\to\infty} \Psi_{\tau_{\mathrm{b}}D,0} = 0$. Therefore, the assumption that $\Psi_{4,\tau_{\mathrm{f}}D} \approx \Psi_{\infty,\tau_{\mathrm{f}}D}$ is incorrect. Using a table based on the plot in Figure 5.1 can cause extreme and incorrect fluctuations in color space, an effect I dub Ψ **clipping**.

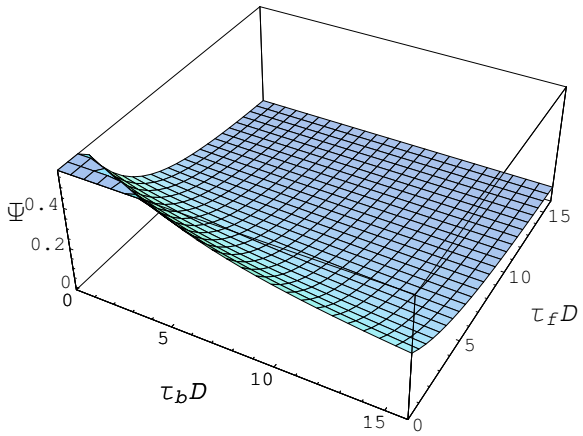


Figure 5.2: Plot of Ψ (Equation 5.8) against $\tau_{\mathrm{b}} D$ and $\tau_{\mathrm{f}} D$ for values in the range $[0, 16]$.

So what can we do about Ψ clipping? One approach is to structure the application such that no integrated ray segment ever has a value of τD that exceeds 4. For many applications, such an approach is perfectly reasonable. After all, providing a larger attenuation does not make the model look more opaque, so why support higher attenuations? On the other hand, if rendering a model with cells varying wildly in size, it may not be possible to pick a range for τ that allows for smaller cells to be rendered opaque while maintaining the constraint that $\tau D < 4$.

Another straightforward approach to eliminate Ψ clipping is to increase the domain of $\Psi_{\tau_{\mathrm{b}} D, \tau_{\mathrm{f}} D}$ until the Ψ clipping effect is below a tolerable threshold. Figure 5.2 demonstrates why this approach will *not* work. In it, we have significantly increased the range over which $\Psi_{\tau_{\mathrm{b}} D, \tau_{\mathrm{f}} D}$ is plotted. The area plotted increased 16 fold, and the surface is far less smooth, thereby necessitating a significant increase in lookup table resolution. Yet $\Psi_{16,0} = 0.3$, which is still significantly greater than $\Psi_{\infty,0}$.

I propose a means of eliminating Ψ clipping by changing the variables we use to index Ψ, much like we did to reduce the dimensionality of the Ψ table as described

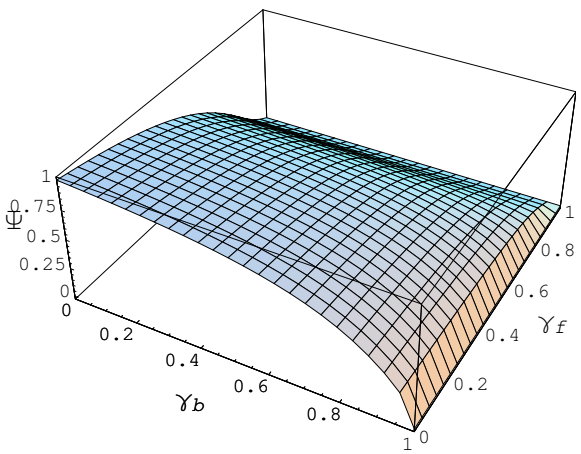


Figure 5.3: Plot of Ψ against γ_{b} and γ_{f} (Equation 5.10).

in Section 5.2.2. First, we define the variable γ as

$$\gamma \equiv \frac{\tau D}{\tau D + 1} \tag{5.9}$$

Solving Equation 5.9 for τD,

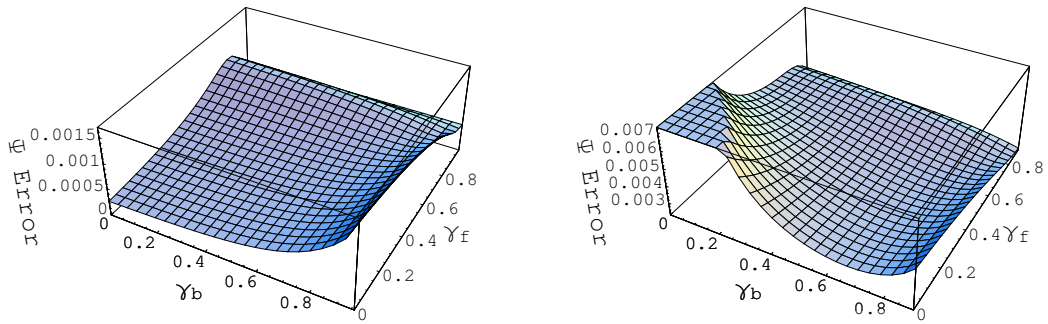
$$\tau D = \gamma/(1-\gamma)$$

and substituting into Equation 5.8, we get the following.

$$\Psi_{\gamma_{\mathrm{b}},\gamma_{\mathrm{f}}} = \int_0^1 e^{-\int_s^1 \left(\frac{\gamma_{\mathrm{b}}}{1-\gamma_{\mathrm{b}}}(1-t)+\frac{\gamma_{\mathrm{f}}}{1-\gamma_{\mathrm{f}}}t\right)dt} ds \tag{5.10}$$

The principle advantage of using γ over τD is that valid values of γ range only over $[0, 1)$. It is therefore possible to store the entire domain of Ψ into a single table. Figure 5.3 shows a plot of $\Psi_{\gamma_{\mathrm{b}},\gamma_{\mathrm{f}}}$ over its entire domain.

Appendix A contains instructions on computing tables of Ψ values with *Mathematica*. Appendix B.4 lists Cg code that uses such a table to compute the volume rendering integral.



(a) Absolute maximum error of Ψ table.

(b) Cumulative maximum Ψ error through 100 cells.

Figure 5.4: Maximum error of Ψ calculation using a lookup table with a resolution of 1024 by 1024. The error is assumed to be caused by adding or subtracting 1/2048 to each γ, which is the maximum distance between the correct value and the nearest value stored in the lookup table.

5.2.4 Resolution of Ψ Table

In this section, we examine the error introduced by using a lookup table to compute Ψ rather than using numerical methods to compute it directly. I arbitrarily picked a resolution of 1024 by 1024 for the table. Using floating-point values for entries, the table takes 4 MB of memory, a large texture but well within the resource limits of today's graphics hardware.

Figure 5.4(a) shows the maximum error introduced by performing a table lookup for any (γ_b, γ_f) pair. The error, measured as the absolute difference between the correct Ψ and the nearest value in the lookup table, seldom reaches above 0.001 of the maximum intensity of the display device, which is below what the human eye is likely to discern. However, these values can be misleading. In practice, we use the value of Ψ to compute the color of a ray segment through just one of many cells.

It is fortunate that the Ψ table error is minimized when values of γ are close

to zero (i.e. the cell is nearly transparent). When the opacity is low, many cells determine the value of the final ray color. When the opacity is high, only the closest few cells contribute to the final ray color. Figure 5.4(b) demonstrates the maximum cumulative error that can occur through 100 cells. As can be seen, although the error for one cell is small when the opacity is lowest, the cumulative error is highest in this region.

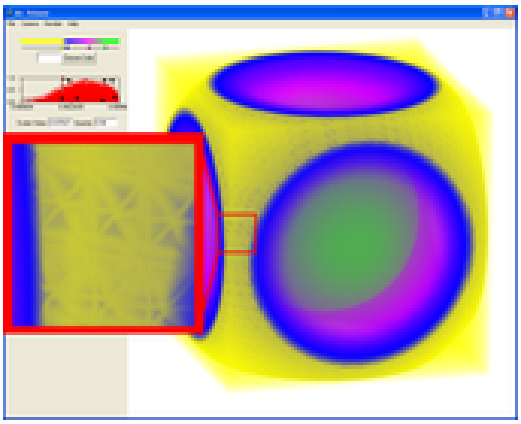
Note, however, that although we see potentially high errors, they occur only in pathological conditions with minimal attenuation and maximum fluctuation in luminance. Such errors are therefore unlikely to occur in practice. We can reduce the potential error by increasing the resolution of the lookup table or performing linear interpolation among values when performing the lookup.

Figure 5.5 compares various approaches for performing ray integration. The approximation that averages both the luminance and the attenuation has obvious errors where the blue is bleeding through the yellow. Partial pre-integration eliminates these errors. The numerical methods from Williams, Max, and Stein [105] give equally good results, but, as we shall see in Chapter 6, this approach takes over an order of magnitude longer than partial pre-integration to compute.

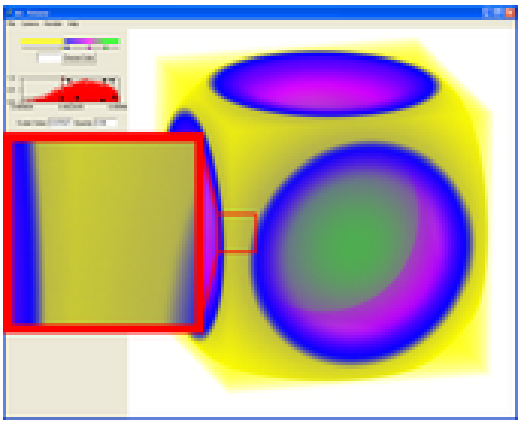
5.3 Linear Interpolation of Opacity

Although interpolating the attenuation parameter (τ) linearly is common, it can lead to problems. As the attenuation changes linearly, the **opacity** (α), the fraction of incoming light occluded by the volume, changes exponentially. Because the observable effect is not changing in proportion to the modified parameter, building a transfer function is difficult.

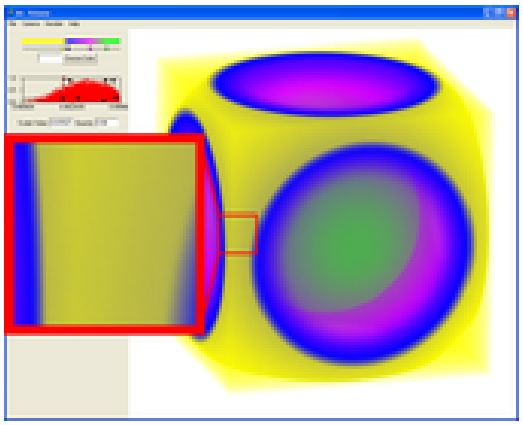
Instead, a preferable option is to parameterize the opacity rather than the atten-



(a) Average Luminance and Attenuation

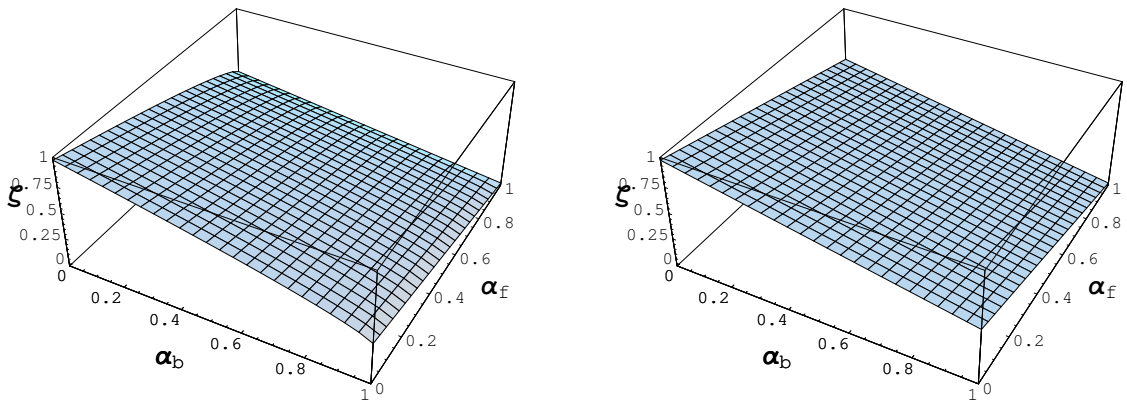


(b) Partial Pre-Integration



(c) Linear Luminance and Attenuation

Figure 5.5: A comparison of ray-integration approaches that linearly interpolate the luminance.



(a) ζ with linearly interpolated opacity.

(b) ζ approximated with average opacity.

Figure 5.6: Approximation of ζ for linearly interpolated opacity (α). Both computations are with unit length segments.

uation of the volume—a subtle but important difference. Wilhelms and van Gelder [102] give a simple relationship between the two.[1]

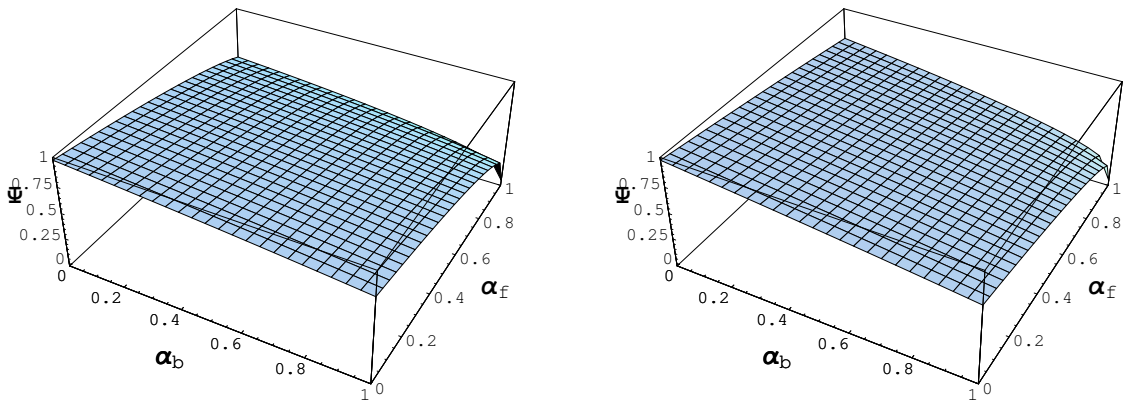
$$\alpha = 1 - e^{-\tau} \tag{5.11}$$

$$\tau = -\ln(1 - \alpha) \tag{5.12}$$

5.3.1 Initial Approximation

Linearly interpolated opacity results in an unwieldy form for ζ. Instead of trying to calculate ζ directly, we can use an approximation similar to that given by Wilhelms and van Gelder [102]. We assume that $\tau(s)$ is constant in Equation 5.3. In this case, $\zeta_{D,\tau} = e^{-D\tau}$. To get a value for τ, we average the opacity ($\alpha(s) \approx \frac{1}{2}(\alpha_b + \alpha_f)$) and then convert that to an attenuation coefficient (via Equation 5.12). Figure 5.6 demonstrates that this approximation is quite close.

[1] Wilhelms' nomenclature is "material opacity" for α and "differential opacity" for τ.



(a) Ψ with linearly interpolated opacity.

(b) Ψ approximated with average opacity.

Figure 5.7: Approximation of Ψ for linearly interpolated opacity (α). Both computations are with unit length segments.

When α varies linearly (τ varies logarithmically), Ψ (Equation 5.4) does not have a closed form. We can approximate Ψ in the same manner as we approximate ζ: by averaging α. If we constrain the opacity and attenuation to be constant, Equation 5.4 reduces to

$$\Psi_{D,\tau} = \frac{1 - e^{-D\tau}}{D\tau} = \frac{1 - \zeta}{D\tau}$$

Figure 5.7 shows us that this approximation is also reasonable.

Appendix B.5 lists Cg code that we can use to perform this volume rendering integral approximation. However, under rare circumstances when the opacity changes drastically over a large cell, errors can become noticeable.

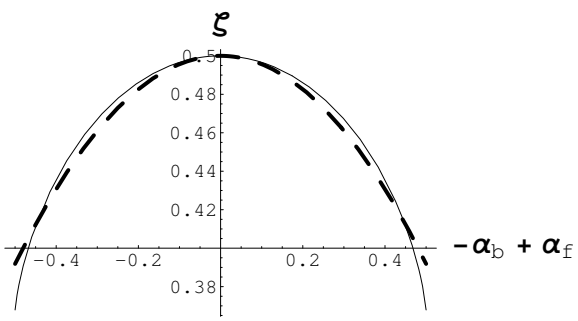


Figure 5.8: Plot of ζ with linearly interpolated α, where $\alpha_\mathrm{b} + \alpha_\mathrm{f} = 1$ and the length of the ray is 1. The solid line is the actual ζ. The dashed line is a parabola we fit over this curve.

5.3.2 Improved Approximation

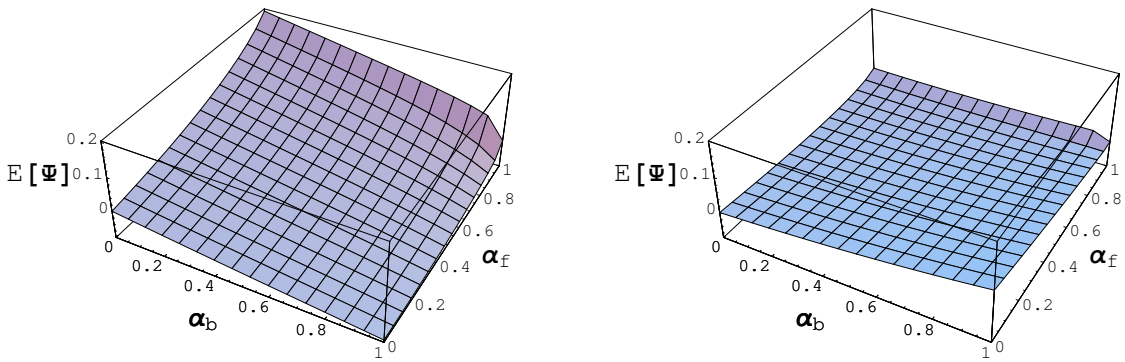
Although the plot in Figure 5.6(a) is mostly flat, like our approximation, there is a tapering at the corners where the two α parameters are maximally different. Figure 5.8 shows the plot of the cross section between these two corners. The result looks like a parabola. We are able to reduce the error significantly by fitting a parabola to this plot and using

$$\alpha(s) \approx \frac{1}{2}\left(\alpha_\mathrm{b} + \alpha_\mathrm{f}\right) + 0.108165\left(\alpha_\mathrm{b} - \alpha_\mathrm{f}\right)^2 \tag{5.13}$$

instead of $\alpha(s) \approx \frac{1}{2}\left(\alpha_\mathrm{b} + \alpha_\mathrm{f}\right)$.

Our approximation of Ψ is trivially correct when $\alpha_\mathrm{b} = \alpha_\mathrm{f}$ but can have noticeable error when the two opacities are significantly different. Because we are averaging the opacity, our approximation yields the same result for any pair of $(\alpha_\mathrm{b}, \alpha_\mathrm{f})$ that satisfies $\alpha_\mathrm{b} + \alpha_\mathrm{f} = c$. However, Ψ should have smaller values when α_f is larger and the front color becomes more predominant than the back color. We can capture this effect by using a weighted sum for $\alpha(s)$ rather than an average. Using

$$\alpha(s) \approx 0.27\alpha_\mathrm{b} + 0.73\alpha_\mathrm{f} \tag{5.14}$$



(a) Error of original approximation of Ψ.

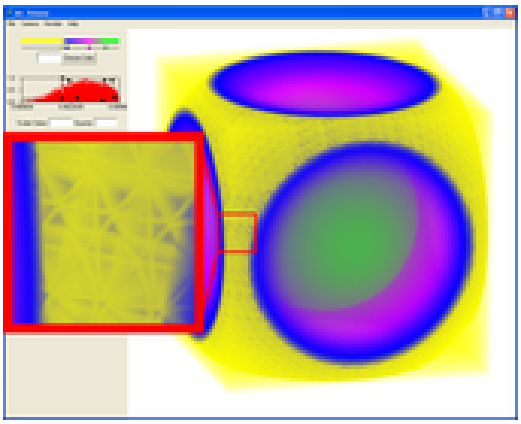
(b) Error of adjusted approximation of Ψ.

Figure 5.9: Plots the error of the Ψ approximations. Both plots give values for unit length segments.

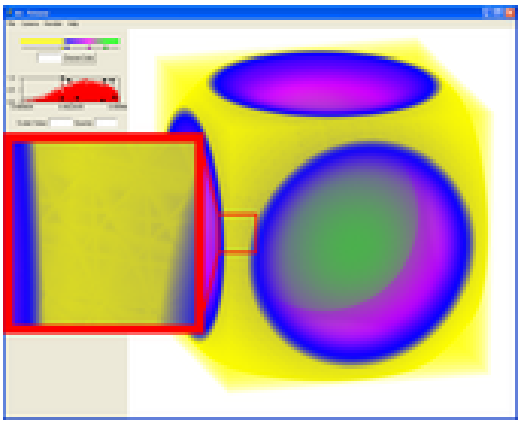
instead of $\alpha(s) \approx \frac{1}{2}(\alpha_b + \alpha_f)$ yields a much lower error, as Figure 5.9 demonstrates. I get this weighting by using *Mathematica* [108] to minimize the l_2-norm of the errors of 3840 samples over the domain of Ψ.

Appendix B.6 lists Cg code that can use to perform this improved volume rendering integral approximation. When compiled with the latest NVIDIA Cg compiler, it requires 11 more instructions than the code listed in Appendix B.6 for the approximation given in Section 5.3.1.

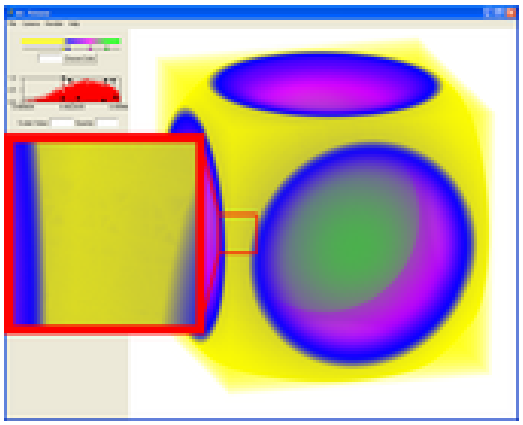
Figure 5.10 compares various approaches for performing ray integration. The approximation that averages both the luminance and the opacity (Figure 5.10(a)) has obvious errors where the blue is bleeding through the yellow. The approximation that averages the opacity but linearly interpolates the color, introduced in Section 5.3.1, (Figure 5.10(b)) is a vast improvement. The same errors still exist, but are vary faint. The approximation for linear luminance and opacity introduced in this section (Figure 5.10(c)) reduces the errors yet again.



(a) Average color and opacity



(b) Linear color, average opacity



(c) Linear color and opacity approximation

Figure 5.10: A comparison of ray-integration approaches that linearly interpolate the luminance.

5.4 Synopsis

An often used approximation for the volume rendering integral is to average the luminance over the ray segment [64, 86, 102]. Although the computation is fast enough to compute in real time, the errors introduced by the approximation are noticeable. A closed form for the volume rendering integral with linearly interpolated luminance and attenuation is known [104, 105], but it takes more than an order of magnitude longer to compute than the approximation, making it suitable only for off-line rendering.

In Section 5.2, I introduced partial pre-integration. Partial pre-integration performs the same linear interpolation on luminance and attenuation as [105], but in a fraction of the time. Partial pre-integration is accurate yet fast enough to use in real time applications.

Rather than linearly interpolate attenuation, linearly interpolating opacity can facilitate the building of transfer functions. There is no closed form of the volume rendering integral for linearly interpolated luminance and opacity, but [102] provides a rough approximation. In Section 5.3, I provide improvements to the approximation that eliminate visual artifacts.

Chapter 6

Results and Comparisons

In this chapter, we explore the performance of the volume rendering methods introduced in Chapter 4 and Chapter 5. We shall analyze the volume rendering on several fronts. First, we examine the speed at which the volume rendering runs. Second, we analyze the accuracy of the volume rendering. That is, we determine how closely our approximations compute the actual volume rendering integrals specified by our model. Third, we examine what effect cell boundaries have on the approximation. We compute the ray integrals over a cell boundary where the color should be consistent and look for variations and discontinuities in the output color that make errors more noticeable.

6.1 Speed

Rendering speeds of a volume rendering system can vary with the volume being rendered, the transfer function being used, the viewing projection, and the image size. In this chapter, I give rendering times for several data sets taken from the

Data Set	Vertices	Tetrahedra
Blunt Fin	40,921	187,395
Oxygen Post	108,300	513,375
Delta Wing	211,680	1,005,675

Table 6.1: Data sets used for testing.

NASA Advanced Supercomputing website[1] and converted to tetrahedra. Table 6.1 lists the volumes used. Figure 6.1 shows sample renderings of the volumes.

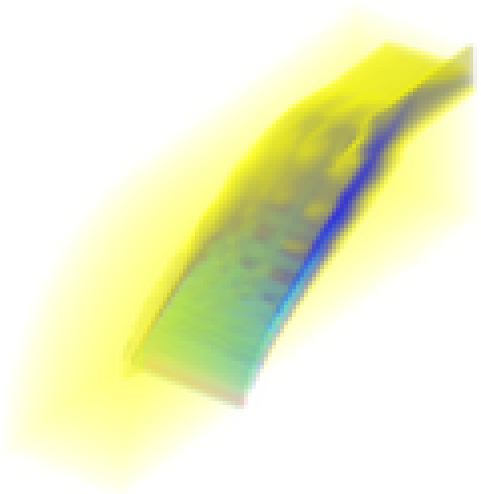
The timings given in this section are for 800 by 800 pixel images unless otherwise specified. The transfer function used for each model is a fixed function with 8 to 10 control points. The renderings rotate the camera around the center of the model. The frame times given are an average of the rendering speed over every frame through the rotation. I performed the tests on a 3.2 GHz Pentium 4 with 2 GB of RAM and a Quadro FX 3000 graphics card. The Quadro graphics card has 256 MB of its own memory and resides on an AGP 8X bus.

6.1.1 Cell Projection

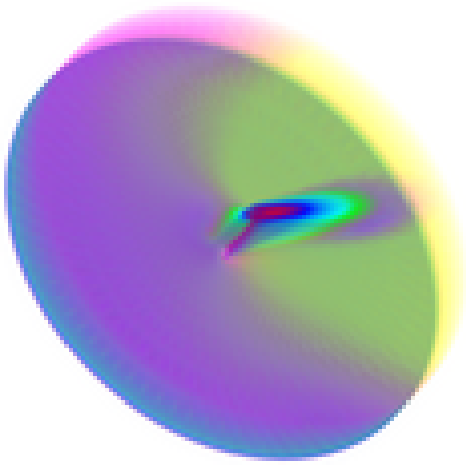
The cell projection that performs adaptive transfer function sampling, described in Section 4.3, clips tetrahedra and renders them multiple times. Before analyzing the rendering rate of this cell-projection method, we must first understand how many more tetrahedra we must render to perform the adaptive transfer function sampling. Table 6.2 gives, for each data set, the number of extra tetrahedra rendered. All the transfer functions selected require the adaptive transfer function sampling method to render about 33% more tetrahedra.

Table 6.3 compares view independent cell projection with the two cell-projection methods defined in Chapter 4. Figure 6.2 summarizes these results in a bar graph.

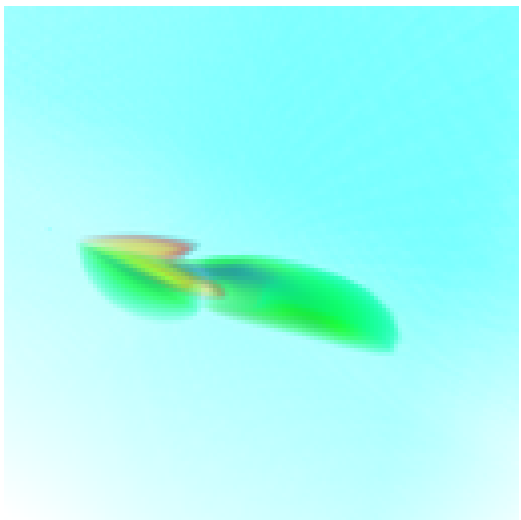
[1] http://www.nas.nasa.gov/Research/Datasets/datasets.html



(a) Blunt Fin



(b) Oxygen Post



(c) Delta Wing

Figure 6.1: Sample data sets.

Data Set	Tetrahedra in Data Set	Tetrahedra Rendered	Growth
Blunt Fin	187,395	249,278	33%
Oxygen Post	513,375	662,625	29%
Delta Wing	1,005,675	1,373,010	36%

Table 6.2: Growth in data sets for adaptive transfer function sampling. This table gives the size of the original data set, the number of tetrahedra rendered by the adaptive transfer function sampling approach, and the growth in the number of tetrahedra rendered.

Model	Cell Projection	Frames/sec	Tetrahedra/sec
Blunt Fin			
	View Independent Cell Projection	3.296	618 K
	Balanced Cell Projection	4.498	843 K
	Adaptive Transfer Function Sampling	1.312	327 K
Oxygen Post			
	View Independent Cell Projection	1.343	690 K
	Balanced Cell Projection	2.275	1168 K
	Adaptive Transfer Function Sampling	0.599	397 K
Delta Wing			
	View Independent Cell Projection	0.721	725 K
	Balanced Cell Projection	1.556	1565 K
	Adaptive Transfer Function Sampling	0.421	578 K

Table 6.3: Running times for various volume rendering cell-projection approaches. Methods printed in blue represent implementations of previous work whereas methods printed in green are introduced in this dissertation.

View Independent Cell Projection is the method developed by Weiler, Kraus, and Ertl [98] and reviewed in Section 4.1. *Balanced Cell Projection* and *Adaptive Transfer Function Sampling* are the methods presented in Sections 4.2 and 4.3, respectively. To highlight the running times of each cell-projection method, I used the least computationally intensive ray integration methods. For *View Independent Cell Projection* and *Balanced Cell Projection*, I used pre-integration to perform ray integration. For *Adaptive Transfer Function Sampling*, I simply averaged the color and opacity of each ray segment.

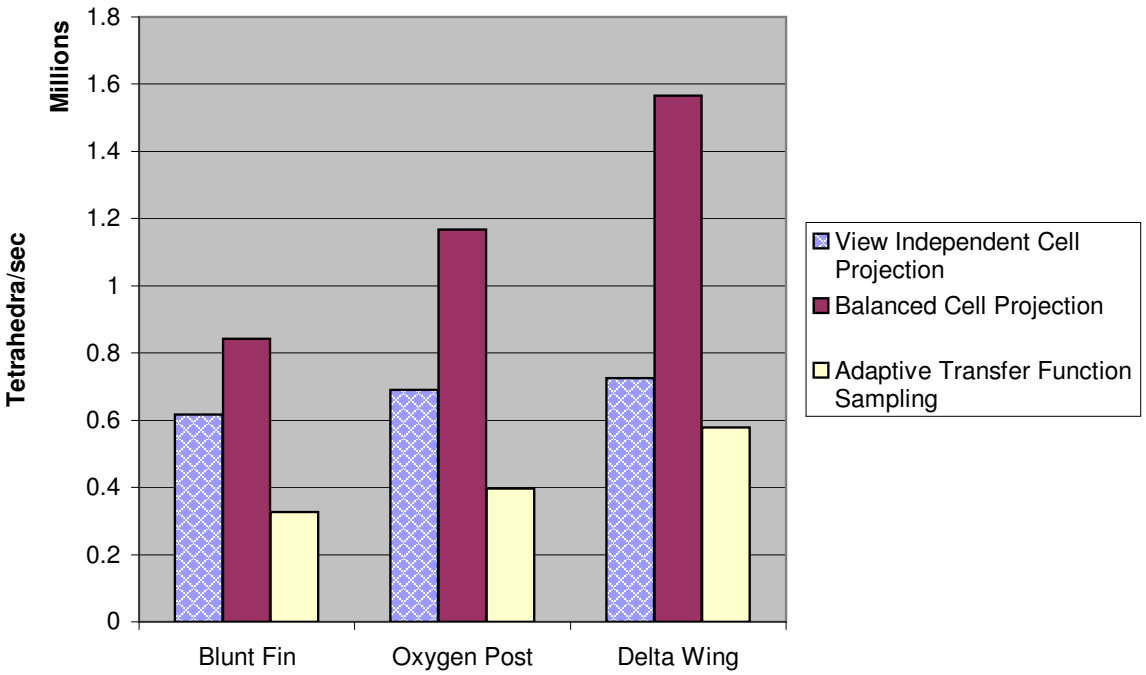


Figure 6.2: Running times for various volume rendering cell-projection approaches. The bars with the hatched fill represent implementations of previous work whereas solid fill represents methods introduced in this dissertation.

My *Balanced Cell Projection* is a modified version of *View Independent Cell Projection*, and the rendering times suggest that these changes do indeed speed up the rendering. *Adaptive Transfer Function Sampling* is the same as *Balanced Cell Projection* with the added ability to clip cells. We expect the added computation to clip cells to affect performance, and the data show that it does. In these ways, the comparative running times verify our preconceived notions of how well these cell-projection methods perform.

However, these results also differ somewhat from what I would expect. I would expect the improvements of the *Balanced Cell Projection* over the *View Independent Cell Projection* to be more dramatic. Furthermore, the penalty of the *Adaptive Transfer Function Sampling* is more costly than I would expect. I believe these results arise from the system being fragment-processing bound. If the fragment

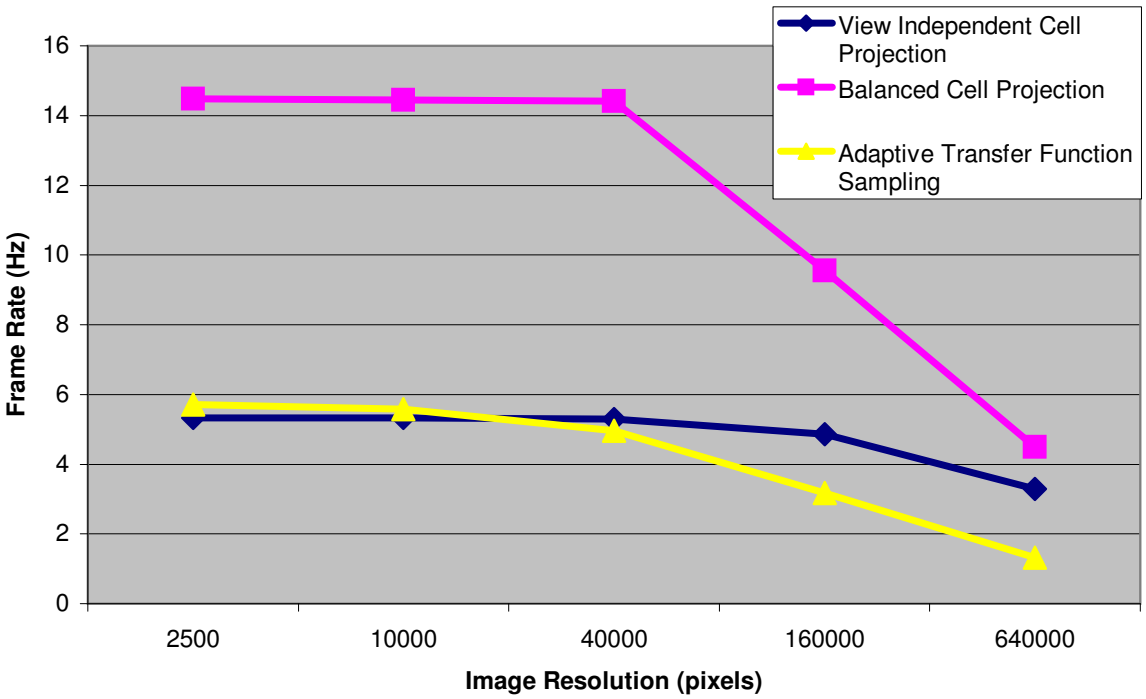


Figure 6.3: Effect of fragment processing on cell projection speed. All readings are taken from the Blunt Fin data set.

processing were the bottleneck, it would diminish improvements in the cell projection. Furthermore, the clipping performed in the *Adaptive Transfer Function Sampling* relies heavily on the fragment processor.

Figure 6.3 shows how the rendering rate changes as the image size (and consequently the number of fragments processed) increases. For all cell-projection methods, the rendering rate holds nearly constant until the image size reaches four thousand pixels. After that, the renderer becomes fragment processor bound and the rendering rate steadily decreases as the image size increases.

Model	Ray Integration	Frames/sec	Tet/sec
Blunt Fin			
	Average Luminance and Attenuation	1.314	328 K
	Linear Luminance and Attenuation	0.080	20 K
	Partial Pre-Integration	0.937	234 K
	Average Luminance and Opacity	1.312	327 K
	Linear Luminance, Average Opacity	1.196	298 K
	Linear Luminance and Opacity Approx	0.938	234 K
Oxygen Post			
	Average Luminance and Attenuation	0.577	382 K
	Linear Luminance and Attenuation	0.028	18 K
	Partial Pre-Integration	0.338	224 K
	Average Luminance and Opacity	0.599	397 K
	Linear Luminance, Average Opacity	0.417	276 K
	Linear Luminance and Opacity Approx	0.337	224 K
Delta Wing			
	Average Luminance and Attenuation	0.407	558 K
	Linear Luminance and Attenuation	0.029	40 K
	Partial Pre-Integration	0.313	430 K
	Average Luminance and Opacity	0.421	578 K
	Linear Luminance, Average Opacity	0.375	515 K
	Linear Luminance and Opacity Approx	0.309	424 K

Table 6.4: Running times for various volume rendering ray integration approaches. Methods printed in blue represent implementations of previous work whereas methods printed in green are introduced in this dissertation.

6.1.2 Ray Integration

Table 6.4 compares the various methods for computing the volume rendering integral that I discussed in this dissertation. Figure 6.4 summarizes these results in a bar graph. *Average Luminance and Attenuation* is the approximation reviewed in Section 3.2.2 and *Linear Luminance and Attenuation* is the full computation of the volume rendering integral reviewed in Section 3.2.3. *Partial Pre-Integration* is the fast computation of linear luminance and attenuation introduced in Section 5.2. *Average Luminance and Opacity* is the same as *Average Luminance and Attenua-*

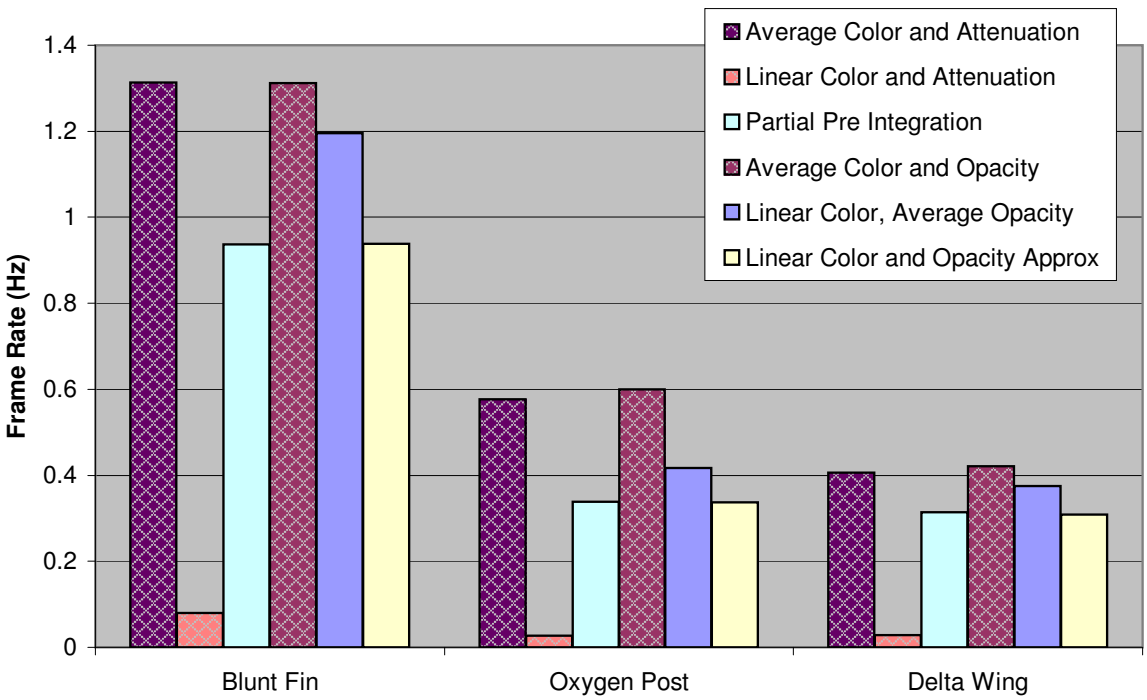


Figure 6.4: Running times for various volume rendering ray integration approaches. The bars with the hatched fill represent implementations of previous work whereas solid fill represents methods introduced in this dissertation.

tion except that the former averages opacity rather than attenuation. This method is similar to that used by Wilhelms and van Gelder [102]. *Linear Luminance, Average Opacity* and *Linear Luminance and Opacity Approx* are the approximations presented in Sections 5.3.1 and 5.3.2, respectively.

Note that I perform the ray integration for all these methods exclusively in the fragment processor and recall from the previous section that the renderer is fragment-processing bound for these tests. Therefore, the comparative rates shown in Table 6.4 are good indicators of the relative performance of the different methods.

The *Average Color and Luminance* approach pioneered by Shirley and Tuchman [86] has one of the fastest frame rates, but, as we see in the following sections, can have large errors caused by color averaging. The *Linear Color and Luminance* com-

putation developed by Williams, Max, and Stein [105] has superior image quality but abysmal rendering rates. In contrast, the *Partial Pre-Integration* method introduced in this dissertation has a rendering speed competitive with the Shirley and Tuchman method yet, as we see in the following sections, its accuracy is competitive with that of the Williams, Max, and Stein method.

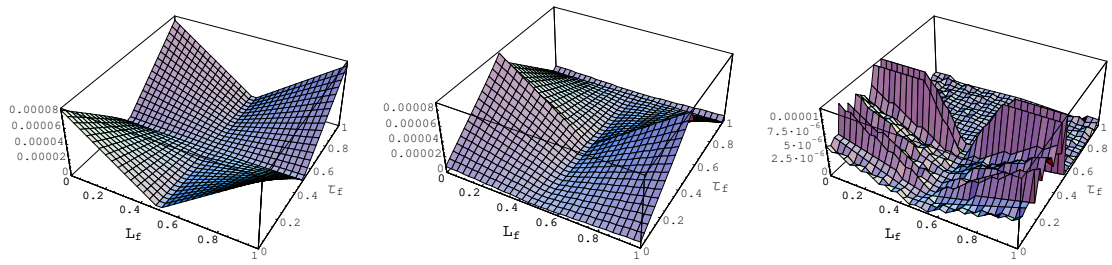
The *Average Color and Opacity* approach used by Wilhelms and van Gelder [102] also has excellent frame rates but poor image quality. Both the *Linear Color, Average Opacity* and *Linear Color and Opacity Approx* methods have competitive frame rates but more accuracy. The *Linear Color, Average Opacity* method is slightly faster, but the *Linear Color and Opacity Approx* is sometimes more accurate.

6.2 Accuracy

In this section, we analyze the accuracy of the various methods for computing the volume rendering integral outlined in this dissertation. I measure the absolute deviation of the light intensity computed with each approximation from the true value of the associated model. Because the basic function of the receptors in the human eye is to measure light intensity, minimizing this deviation is important for good image quality. However, the human visual system is a complicated structure that can adapt well to changes in intensity, so the absolute deviation may not be proportional to the actual error perceived. We shall consider perceptive errors in the following section.

To quantify the error, I solve the volume rendering integral for a set of parameters offline using the numerical solving capabilities of *Mathematica* [108] and compare those to computations performed on the actual graphics card. I used a Quadro FX 3000 graphics card for the GPU calculations.[2]

[2]I used the same hardware to produce all the results presented in this chapter. All ray integration calculations are performed on a full 32-bit per channel GPU. However, the



(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

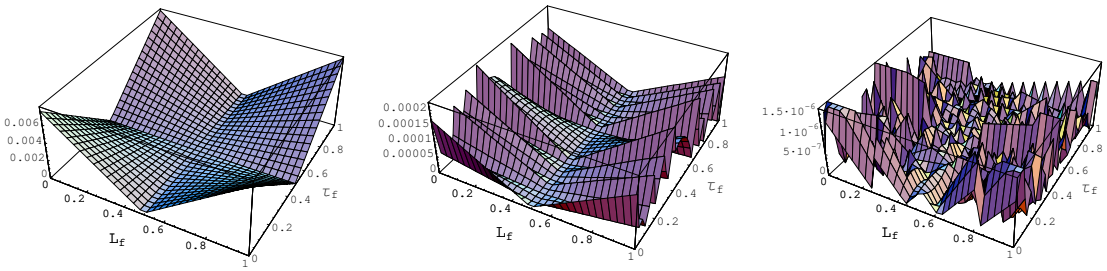
(c) Linear Luminance and Attenuation

Figure 6.5: Error for various approximations to the volume rendering integral. For all plots, $L_b = 1 - L_f$, $\tau_b = 1 - \tau_f$, and $D = 0.001$.

The volume rendering integral with linearly varying properties relies on five parameters. Rather than present an exhaustive report of errors on all combinations of these parameters, I show the combinations where the error is the greatest. All the approximations reviewed or presented in this dissertation are correct when the luminance and attenuation parameters are constant, whereas the errors of these approximations are maximal when the color and attenuation are both changing rapidly.

Figure 6.5 provides plots of the error of the *Average Luminance and Attenuation* and *Partial Pre-Integration* approximations, as well as the brute-force, on-card numerical method of *Linear Luminance and Attenuation*. All values in Figure 6.5 are computed for a ray segment length of $D = 0.001$. Figures 6.6, 6.7, and 6.8 hold similar plots for errors when the ray segment length is 0.1, 1, and 100, respectively. In all plots, the error is the absolute difference between the result of the approximation method and the value computed with the high precision numerical methods available

images produced for the results in Section 6.1 were written to an 8-bit per channel frame buffer whereas the values presented in Sections 6.2 and 6.3 are taken as 32-bit values. I used the 8-bit frame buffer because the color blending required for the image generation is not available for the 32-bit frame buffer. I used 32-bit values for Sections 6.2 and 6.3 to minimize quantization errors, which are orthogonal to the errors intrinsic to the approximations.

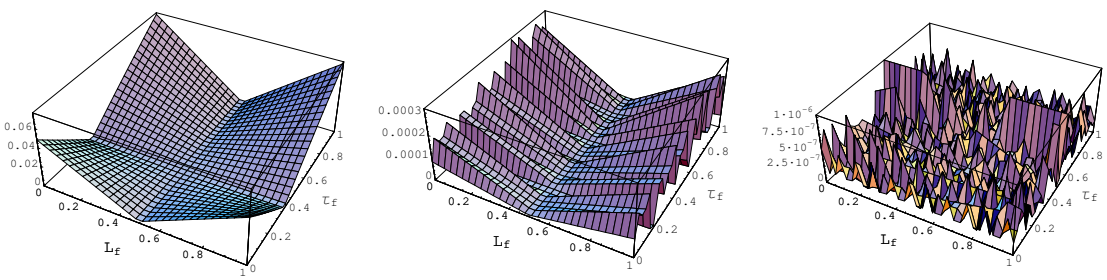


(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

(c) Linear Luminance and Attenuation

Figure 6.6: Error for various approximations to the volume rendering integral with $D = 0.1$.



(a) Average Luminance and Attenuation

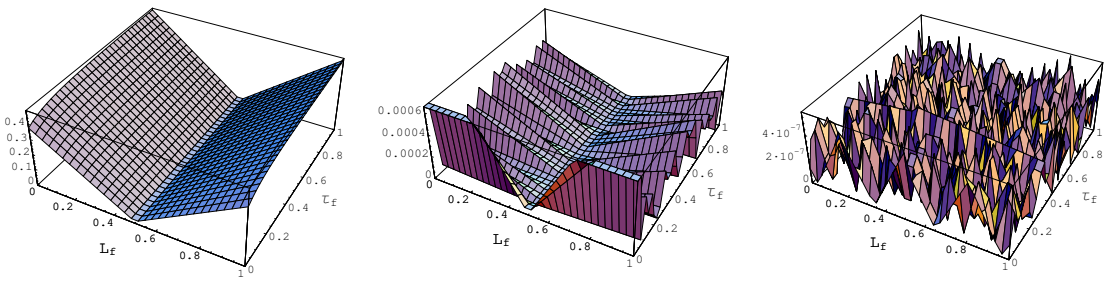
(b) Partial Pre-Integration

(c) Linear Luminance and Attenuation

Figure 6.7: Error for various approximations to the volume rendering integral with $D = 1$.

with the *Mathematica* software package [108]. Assuming, without loss of generality, that valid values of intensity are in the range $[0, 1]$, 0 being no light and 1 being the maximum output intensity of the display device, we can take the error plotted as the fraction of the display's intensity range for which the value is incorrect.

For all ranges of ray segment length, *Average Luminance and Attenuation* has the poorest performance. Furthermore, the error gets larger with longer ray segments.



(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

(c) Linear Luminance and Attenuation

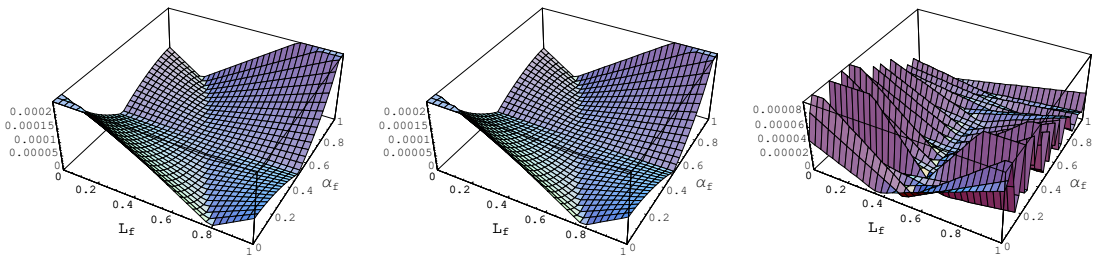
Figure 6.8: Error for various approximations to the volume rendering integral with $D = 100$.

As the segment length gets large, the maximum error approaches one-half of the full color range.

Partial Pre-Integration has consistently lower error overall. Furthermore, the error grows slowly with the length of the ray segment. Its accuracy is limited only by the quantization errors of its lookup table.

The accuracy of *Linear Luminance and Attenuation* from [105] is close to the precision of the 32-bit floating-point variables on which it is calculated. This accuracy is marginally higher than that of *Partial Pre-Integration*. However, visual discrepancies between the two methods are unlikely, and, as we saw in Section 6.1, the *Linear Luminance and Attenuation* method takes over ten times as long as *Partial Pre-Integration*.

Figure 6.9 provides plots of the accuracy the *Average Luminance and Opacity*, *Linear Luminance, Average Opacity*, and *Linear Luminance and Opacity Approx* approximations. All values in Figure 6.9 are computed for a ray segment length of $D = 0.001$. Figures 6.10 and 6.11 hold similar plots for errors when the ray segment length is 0.1 and 1, respectively.

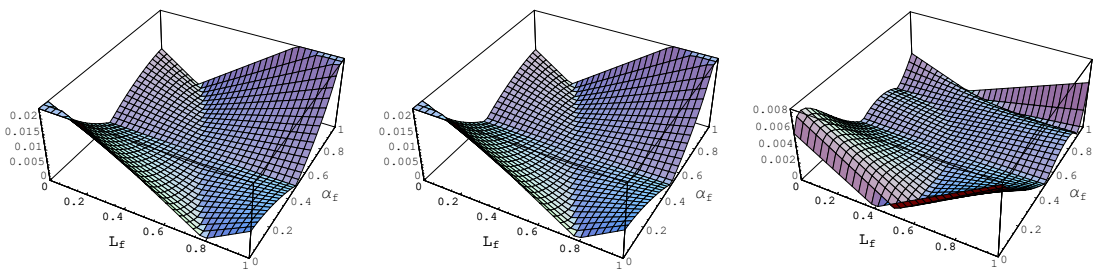


(a) Average Luminance and Opacity

(b) Linear Luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.9: Error for various approximations to the volume rendering integral. For all plots, $L_\mathrm{b} = 1 - L_\mathrm{f}$, $\tau_\mathrm{b} = 1 - \tau_\mathrm{f}$, and $D = 0.001$.



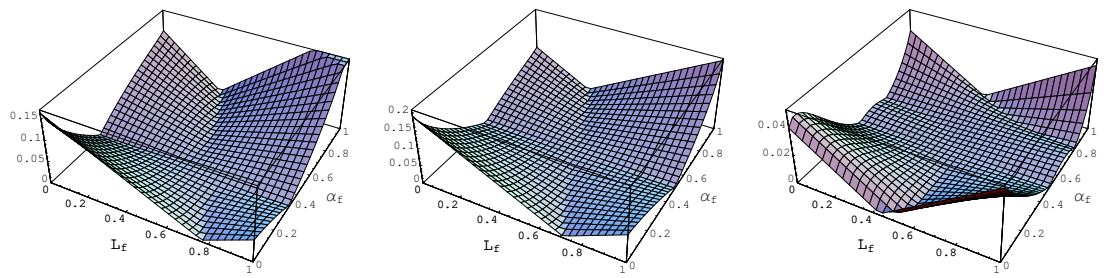
(a) Average Luminance and Opacity

(b) Linear Luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.10: Error for various approximations to the volume rendering integral with $D = 0.1$.

The accuracy of *Average Luminance and Opacity* is poor. The accuracy of *Linear Luminance, Average Opacity* is no higher in the worst case. As before, the error gets larger with longer ray segments. The *Linear Luminance and Opacity* method improves the error by almost an order of magnitude.



(a) Average Luminance and Opacity

(b) Linear Luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.11: Error for various approximations to the volume rendering integral with $D = 1$.

6.3 Cell Boundary Smoothness

In this section, we analyze how the various methods for computing the volume rendering integral outlined in this dissertation behave across cell boundaries. As in the previous section, I solve the volume rendering integral for a set of parameters offline using the numerical solving capabilities of *Mathematica* [108] and compare those to computations performed on the actual graphics card. I used a Quadro FX 3000 graphics card for the GPU calculations.

The difference between the measurements in this section and those in the previous section is that those in this section take into account spatial effects. Viewers are unlikely to notice differences in color if they are all uniform. After all, there are uniform differences in color with different display media or with the parameters of the medium (for example the paper type in a printer or adjustment controls on a monitor). Furthermore, the response of a human's visual system is constantly changing with its environment.

Although the human visual system readily adjusts to uniform changes in light

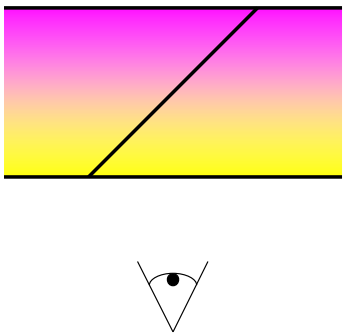


Figure 6.12: Model used to study Mach bands caused by approximation errors.

intensity, it is sensitive to changes in light intensity across its field of vision. This sensitivity is critical for segmenting a visual scene and helps us identify the size, shape, and orientation of objects. If the errors introduced by our approximations are not uniform, they may become noticeable. Therefore, in this section we analyze how the error may change spatially within the image.

The light receptors in the human eye are clustered together into **ganglion cells** [26]. The receptors in the center of each cell have a positive response to incoming light whereas those toward the edge have a negative response. When aimed at a constant field of light, the positive and negative receptors cancel each other out. When aimed at a varying field of light, the positive and negative receptors may contribute in the same way, which enhances the effect of the change. These enhancements generated by our visual system are **Mach bands**.

Figure 6.12 shows the model I used to study how approximation errors may cause fluctuations in colors across the viewing plane, which could induce the human visual system to create mach bands. The model is such that from the viewpoint the volume has a uniform length. The front and back faces of the model each have constant volume parameters. The color of the volume should be constant from the viewpoint, but approximations may cause the color to vary. For each approximation, I plot the output intensity across the face. I also plot the convolution of the light

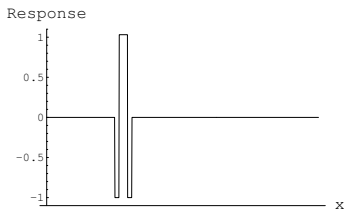
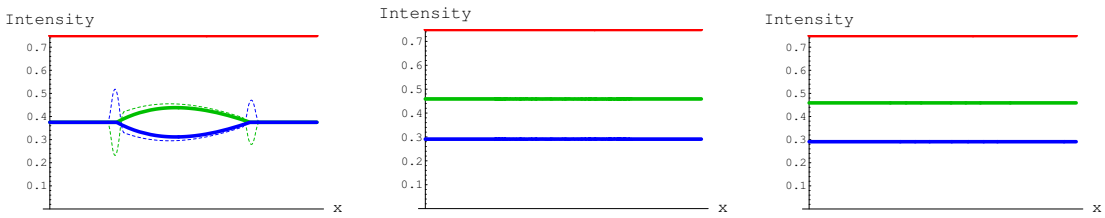


Figure 6.13: Ganglion receptor response.



(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

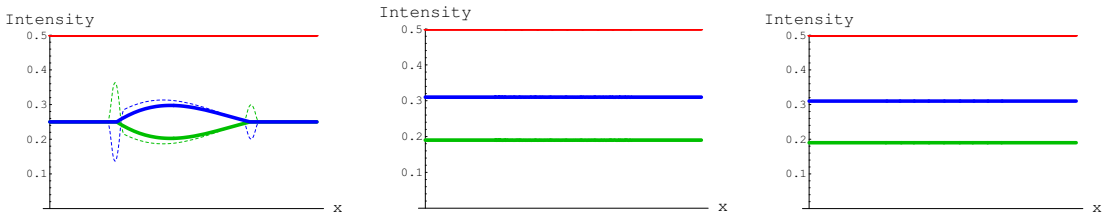
(c) Linear Luminance and Attenuation

Figure 6.14: Color output from model shown in Figure 6.12. The attenuation is constant. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.

intensity with an example ganglion receptor response function shown in Figure 6.13.

Figures 6.14, 6.15, and 6.16 show the output of our model with various ray integration methods that linearly interpolate attenuation and with various volume parameter combinations. The *Average Luminance and Attenuation* model has noticeable spikes when convolved with the ganglion response. The other two models have no noticeable fluctuations across the visual field.

Figures 6.17, 6.18, and 6.19 show the output of our model with various ray integration methods that linearly interpolate opacity and with various volume parameter combinations. The *Average Luminance and Opacity* and *Linear Luminance, Average Opacity Approx* models both have significant spikes when convolved with the

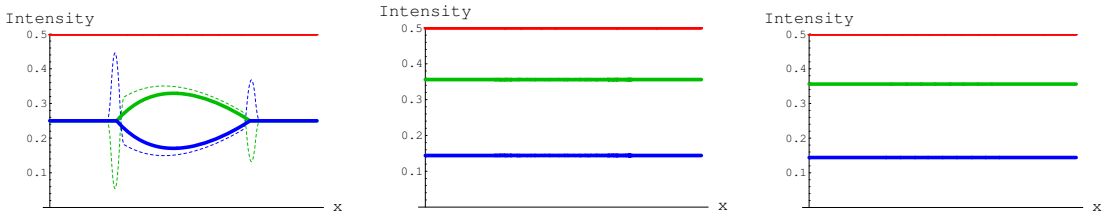


(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

(c) Linear Luminance and Attenuation

Figure 6.15: Cell boundaries with large attenuation in the back. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.

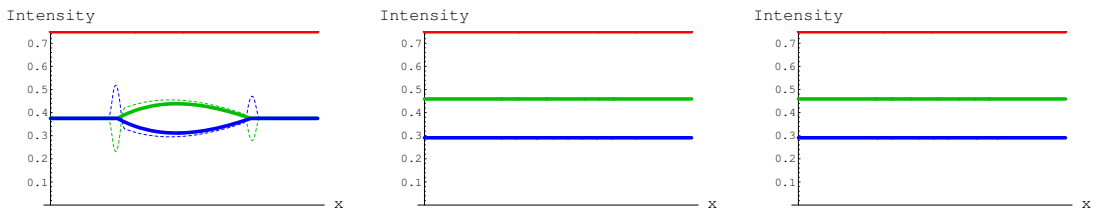


(a) Average Luminance and Attenuation

(b) Partial Pre-Integration

(c) Linear Luminance and Attenuation

Figure 6.16: Cell boundaries with large attenuation in the front. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.

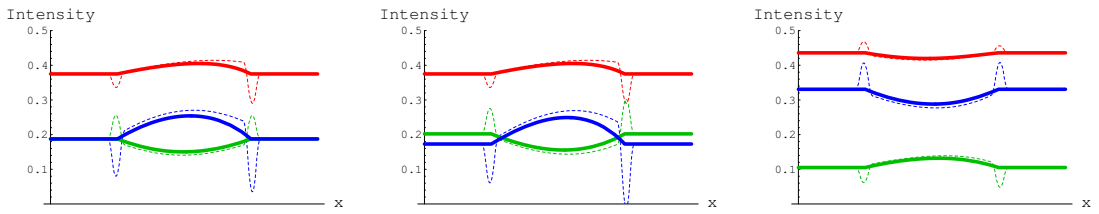


(a) Average Luminance and Opacity

(b) Linear luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.17: Color output from model shown in Figure 6.12. The opacity is constant. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.

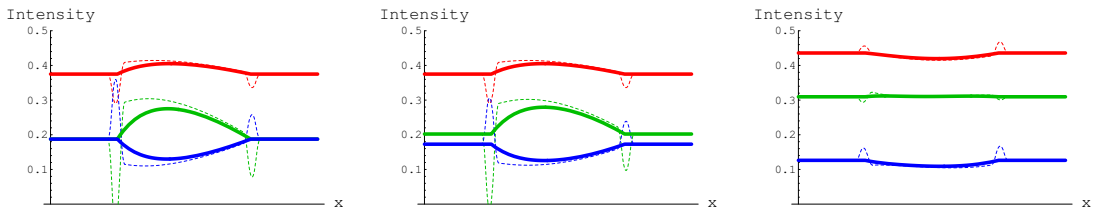


(a) Average Luminance and Opacity

(b) Linear luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.18: Cell boundaries with large opacity in the back. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.



(a) Average Luminance and Opacity

(b) Linear luminance, Average Opacity

(c) Linear Luminance and Opacity Approx

Figure 6.19: Cell boundaries with large opacity in the front. In the red plot, the luminance is constant. The luminance is full in the front and zero in the back for the green plot and vice versa for the blue plot. The dashed plots show the convolution with the ganglion response of Figure 6.13.

ganglion response. In contrast, the *Linear Luminance and Opacity* model has little fluctuation, even when convolved with the ganglion response.

Although I have addressed spatial effects of image error in this chapter, the errors I quantify may still not be indicative of the error perceived. Ganglion cells and Mach bands are but a small part of the human visual system. In fact, there is still much we do not understand about the human visual system. To get a true measure of how well humans are able to detect the errors discussed in this chapter, one must generate a set of images from different ray integration methods and compare them either by human experiment or with image quality metrics based off human experiment [23, 65, 96].

Chapter 7

Conclusions

The goal of this dissertation is to improve the state of the art in unstructured mesh volume rendering. Over the past decade, there has been research to perform volume rendering that is either fast [86, 98, 102, 110] or accurate [92, 105], but not both. This dissertation presented algorithms that are as fast as the former but as accurate as the latter.

I started with the *View Independent Cell Projection* algorithm [98, 100]. The speed of the algorithm was obtained by taking advantage of recent improvements in accelerated graphics hardware. I have made several improvements to this algorithm and demonstrated their effectiveness.

The original *View Independent Cell Projection* implementation used *Pre-Integration* [81] to perform its color computations. *Pre-Integration* had many advantages. Because the heavy ray integrations were done off line, *Pre-Integration* could yield high frame rates. Given a large enough table, *Pre-Integration* eliminated errors from sampling the transfer function. In addition, *Pre-Integration* could potentially support any type of ray integration method.

However, *Pre-Integration* had several design limitations. The pre-integration table was built specifically for a given transfer function. Thus, the table had to be rebuilt every time the transfer function changed, which, during a practical application, was often. Furthermore, the accuracy of *Pre-Integration* relied heavily on the size of the table used and ray integrations performed. However, larger tables and more accurate ray-integration methods would slow down the table building. *Pre-Integration* worked only with 1D transfer functions, whereas higher dimensional transfer functions could more effectively highlight features in the volume [46, 47, 50, 94]. *Pre-Integration* was incapable also of performing many non-photorealistic feature highlighting techniques [21, 43, 44]. For these reasons, my algorithms did not rely on *Pre-Integration*.

To avoid aliasing without *Pre-Integration*, I implemented *Adaptive Transfer Function Sampling*. Although it improved image quality, the *Adaptive Transfer Function Sampling* negatively affected the rendering speed. However, even with *Adaptive Transfer Function Sampling*, my algorithm ran at about the same speed as *View Independent Cell Projection*.

My algorithms performed ray integration in the graphics card during rendering. To do this, I devised new ray-integration methods that were both fast and accurate. These ray-integration methods are for linearly varying volume parameters. I provided methods for both linearly varying volume density, which has a closed form, and linearly varying observable opacity, which does not have a closed form. I have shown that these ray-integration methods are competitive with both the speed of previous fast approximations [86, 102] and the accuracy of slow approximations [105].

Sandia National Laboratories is currently using the algorithms introduced in this dissertation for its scientific visualization needs. Furthermore, the code is integrated into the *Visualization Toolkit* (VTK) [84] and will soon be incorporated into *ParaView* [57], a fully featured, open-source scientific visualization package.

The work in this dissertation may take several directions. First, although I have contrasted the ray integrations presented in this dissertation with the approach of *Pre-Integration*, the two approaches could be complementary. The entries in the pre-integration table are filled with values computed with ray integration. We could use the ray-integration methods presented in this dissertation to increase both the accuracy of the entries in the table and the speed with which they are calculated.

Second, although my volume rendering implementations currently support only 1D transfer functions, there are no fundamental limitations preventing the use of transfer functions of two or more dimensions so long as they are still piecewise linear. However, the representation of a piecewise linear function in two or more dimensions can be problematic, and tools for building multidimensional transfer functions are still being developed.

Third, although the ray integration methods introduced in this dissertation are valid only for piecewise linear approximations, we can approximate higher orders of color variation along the ray with a piecewise linear function without any noticeable visual artifacts. However, for maximal speed, we need to minimize the number of segments used in the piecewise linear approximation. The minimal segmentation needs to be investigated.

Appendix A

Computing Pre-Integration Tables with *Mathematica*

The following is a script that we can use to make Ψ tables with *Mathematica*. The partial pre-integration method introduced in Section 5.2 uses the Ψ table generated. The script builds a 1024 by 1024 table. You may change the size of the table by changing the step size in the last line.

```
Ψgeneral = (1 / D) Integrate[Exp[-Integrate[τ[t], {t, s, D}]], {s, 0, D}];

Ψbackopaque = FullSimplify[Ψgeneral /. {τ[t_] → (1 - t / D) τbackD / D + t / D τfrontD / D};];

Ψfrontopaque = FullSimplify[
   Ψbackopaque //. {1 / Sqrt[τbackD - τfrontD] → 1 / (I Sqrt[τfrontD - τbackD])};];

Ψconst = Simplify[Ψgeneral /. {τ[t_] → τD / D};];

Simplify[Ψbackopaque /. Erf[x_] → 1 - u[x] Exp[-x * x + p[u[x]]];];

% /. {u[x_] → 1 / (1 + x / 2)};

Ψbackopaquep = % /. p[x_] → -1.26551223 + x (1.00002368 +
        x (0.37409196 + x (0.09678418 + x (-0.18628806 + x (0.27886807 + x (-1.13520398 +
                        x (1.48851587 + x (-0.82215223 + x 0.17087277)))))));

Ψfrontopaquep = Ψfrontopaque;

Ψconstp = If[τD == 0, Evaluate[Limit[Ψconst, τD → 0]], Evaluate[Ψconst]];

Ψprecise = MapAll[Evaluate, Function[{τbackD, τfrontD}, If[τbackD == τfrontD,
     Ψconstp /. τD → τfrontD, If[τbackD > τfrontD, Ψbackopaquep, Ψfrontopaquep]]]];

ReleaseHold[Hold[Table[Abs[N[Ψprecise[γback / (1 - γback), γfront / (1 - γfront)]]],
     {γfront, 0, 1 - step, step}, {γback, 0, 1 - step, step}]] //.
   {step → N[1 / 1024]}] >> "PsiGammaTable.dat"
```

Appendix B

Vertex and Fragment Programs

This appendix lists the vertex and fragment programs referenced throughout this dissertation. All programs use the Cg shader language.

B.1 Clipped Tetrahedron Projection Vertex Program

This listing is a vertex program that is part of the tetrahedra projection with adaptive transfer function sampling. Section 4.3 discusses the algorithm in its entirety.

```
struct rayseg {
  float4 position       : POSITION;  /* Position of front face. */
  float4 distances      : TEXCOORD0; /* Distance of front face to each
                                        face in direction of view
vector. */
  float3 isovalues      : TEXCOORD1; /* x and y are Color lookups for
                                        scalar values of where tetrahedra
                                        (ray segment) is clipped.  x value
                                        is closer to viewer.  z is the
                                        distance between the two isoplanes
```

```
                                        in the view direction. */
  float frontinterp     : TEXCOORD2; /* Interpolates the color of the
                                        front face from the front isovalue
to the back. */
  float4 backinterp     : TEXCOORD3; /* Interpolates the color of each
                                        face from the back isovalue to the
                                        front. */
};

float4 selectmask[4] = {
  {1, 0, 0, 0},
  {0, 1, 0, 0},
  {0, 0, 1, 0},
  {0, 0, 0, 1}
};
float4 invselectmask[4] = {
  {0, 1, 1, 1},
  {1, 0, 1, 1},
  {1, 1, 0, 1},
  {1, 1, 1, 0}
};

rayseg mainvert(float4 position,        /* Position of the vertex. */
                float distance,         /* Distance from the vertex to the
                                           to opposite face in the view
                                           direction (negative if opposite
                                           face is closer to viewpoint).*/
                float2 isovalues,       /* Texture lookup indices for
                                           clipping isovalues.  The x
   index is closer to the
   viewpoint. */
                float2 interpolants,    /* Used to interpolate the actual
                                           colors of the front and back
                                           scalars. */
                float vertNum,
                uniform float4x4 ModelViewProj)
{
  rayseg output;

  output.position = mul(ModelViewProj, position);

  output.distances = selectmask[vertNum]*distance;
```

```
  output.frontinterp = interpolants.x;

  /* Note that we invert the interpolation so that the back scalar is
     interpolated from the back isoplane to the front isoplane. */
  output.backinterp = 1 - (  invselectmask[vertNum]*interpolants.x
                           + selectmask[vertNum]*interpolants.y );

  output.isovalues.xy = isovalues;
  /* Compute distance between isoplanes. */
  if (interpolants.x != interpolants.y)
    {
    output.isovalues.z = distance/(interpolants.y-interpolants.x);
    }
  else
    {
    /* Special case when front and back scalars are equal: distance
       between planes does not matter. */
    output.isovalues.z = 1.0e30;
    }

  return output;
}
```

B.2 Fragment Program for Clipped Tetrahedron Projection

This listing is a fragment program that is part of the tetrahedra projection with adaptive transfer function sampling. Section 4.3 discusses the algorithm in its entirety.

The following program relies on the function `IntegrateRay`, which is not defined. Instead, subsequent sections provide various algorithms for `IntegrateRay`.

```
struct rayseg {
  float4 position       : POSITION;  /* Position of front face. */
```

```
  float4 distances      : TEXCOORD0; /* Distance of front face to each
                                        face in direction of view
vector. */
  float3 isovalues      : TEXCOORD1; /* x and y are Color lookups for
                                        scalar values of where tetrahedra
                                        (ray segment) is clipped.  x value
                                        is closer to viewer.  z is the
                                        distance between the two isoplanes
                                        in the view direction. */
  float frontinterp     : TEXCOORD2; /* Interpolates the color of the
                                        front face from the front isovalue
to the back. */
  float4 backinterp     : TEXCOORD3; /* Interpolates the color of each
                                        face from the back isovalue to the
                                        front. */
};

float4 IntegrateRay(in float4 BackColor, in float4 FrontColor,
                    in float Length);

float4 mainfrag(rayseg input,
                uniform sampler1D TransferFunction,
                uniform float LengthMultiply) : COLOR
{
  float4 mask;

  /* Make mask be 1 for all distances <= 0. */
  mask = (float4)(input.distances <= 0);

  /* Make all these entries larger so that we do not select them. */
  float4 tmp1 = input.distances + mask*1.0e38;

  float2 tmp2 = min(tmp1.xy, tmp1.zw);
  /* distance is actual distance from front to back of ray segment. */
  float distance = min(tmp2.x, tmp2.y);

  /* Make mask be 1 for minimum depth. */
  mask = (float4)(tmp1 == distance);

  float2 interpolants;
  interpolants.x = input.frontinterp;
  interpolants.y = dot(mask, input.backinterp);
```

```
  /* If either interpolation variable is greater than 1, the segment is
     completely outside the iso range. */
  discard (interpolants > 1);

  /* Remove any "empty space" from the distance. */
  distance -= dot(float2(1,1),
                  input.isovalues.z*max(-interpolants, float2(0,0)));

  float4 isocolorFront = tex1D(TransferFunction, input.isovalues.x);
  float4 isocolorBack = tex1D(TransferFunction, input.isovalues.y);

  /* If either distance is negative, it means that face is in between the
     two isosurfaces.  We have to interpolate the actual scalar value in
     this case.  It is the expected case that we have to interpolate at
     least one value. */
  interpolants = max(interpolants, float2(0,0));
  float4 colorFront =lerp(isocolorFront, isocolorBack, interpolants.xxxx);
  float4 colorBack = lerp(isocolorBack, isocolorFront, interpolants.yyyy);

  return IntegrateRay(colorBack, colorFront, distance*LengthMultiply);
}
```

B.3 Volume Rendering Integral with Linear Attenuation and Luminance

This listing is a function that will, given a pair input colors (with the alpha components set to the attenuation parameter τ), compute the volume rendering integral with linear interpolation for the luminance and attenuation. Section 3.2.3 contains the details for the mathematics. The program outputs a fragment that must be properly blended with the image in the frame buffer using the Porter and Duff [74] over operator. See Section 2.2.3 for details on blending.

```
/* Forward declarations. */
float Psi(in float taub, in float tauf, in float length);
```

```
float erf(in float x);
float erfi(in float x);
float dawson(in float x);
float erf_fitting_function(in float u);

float4 IntegrateRay(in float4 BackColor, in float4 FrontColor,
                    in float Length)
{
    float Y = Psi(BackColor.a, FrontColor.a, Length);
    float zeta = exp(-Length*0.5*(BackColor.a+FrontColor.a));

    float4 OutColor;
    OutColor.rgb = FrontColor.rgb*(1-Y) + BackColor.rgb*(Y-zeta);
    OutColor.a = (1-zeta);
    return OutColor;
}

#define M_SQRTPI        1.77245385090551602792981
#define M_SQRT1_2       0.70710678118654752440
#define M_2_SQRTPI      1.12837916709551257390
#define M_1_SQRTPI      (0.5*M_2_SQRTPI)

float Psi(in float taub, in float tauf, in float length)
{
    float difftau = taub - tauf;
    bool useHomoTau = ((difftau > -0.0001) && (difftau < 0.0001));
    bool useErf = difftau > 0;

    float Y;

    if (!useHomoTau) {
        float invsqrt2lengthdifftau = 1/sqrt(2*length*abs(difftau));
        float t = length*invsqrt2lengthdifftau;
        float frontterm = t*tauf;
        float backterm = t*taub;
        float expterm = exp(frontterm*frontterm-backterm*backterm);
        if (useErf) {
          /* Back more opaque. */
            float u = 1/(1+0.5*frontterm);
            Y = u*exp(erf_fitting_function(u));
            u = 1/(1+0.5*backterm);
            Y += -expterm*u*exp(erf_fitting_function(u));
            Y *= M_SQRTPI;
```

```
        } else {
          /* Front more opaque. */
            expterm = 1/expterm;
            Y = 2*(dawson(frontterm) - expterm*dawson(backterm));
        }
        Y *= invsqrt2lengthdifftau;
    } else {
        float tauD = taub*length;
        Y = (1 - exp(-tauD))/tauD;
    }

    return Y;
}

float erf(in float x)
{
  /* Compute as described in Numerical Recipes in C++ by Press, et al. */
/*     x = abs(x);        In this application, x should always be <= 0. */
    float u = 1/(1 + 0.5*x);
    float ans = u*exp(-x*x + erf_fitting_function(u));
/*     return (x >= 0 ? 1 - ans : ans - 1);    x should always be <= 0. */
    return 1 - ans;
}

float erf_fitting_function(in float u)
{
    return
        - 1.26551223 + u*(1.00002368 + u*(0.37409196 + u*(0.09678418 +
        u*(-0.18628806 + u*(0.27886807 + u*(-1.13520398 + u*(1.48851587 +
        u*(-0.82215223 + u*0.17087277)))))));
}

float erfi(in float x)
{
    return M_2_SQRTPI*exp(x*x)*dawson(x);
}

/* Compute Dawson's integral as described in Numerical Recipes in C++ by
   Press, et al. */
#define H 0.4
#define NMAX 6
float dawson_constant0 = 0.852144;
float dawson_constant1 = 0.236928;
```

```
float dawson_constant2 = 0.0183156;
float dawson_constant3 = 0.000393669;
float dawson_constant4 = 2.35258e-6;
float dawson_constant5 = 3.90894e-9;
float dawson(in float x)
{
    float result;
    if (x > 0.2) {
/*      x = abs(x);       In this application, x should always be <= 0. */
        int n0 = 2*floor((0.5/H)*x + 0.5);
        float xp = x - (float)n0*H;
        float e1 = exp((2*H)*xp);
        float e2 = e1*e1;
        float d1 = n0 + 1;
        float d2 = d1 - 2;
        float sum = 0;
        sum = dawson_constant0*(e1/d1 + 1/(d2*e1));
        d1 += 2;  d2 -= 2;  e1 *= e2;
        sum += dawson_constant1*(e1/d1 + 1/(d2*e1));
        d1 += 2;  d2 -= 2;  e1 *= e2;
        sum += dawson_constant2*(e1/d1 + 1/(d2*e1));
        d1 += 2;  d2 -= 2;  e1 *= e2;
        sum += dawson_constant3*(e1/d1 + 1/(d2*e1));
        d1 += 2;  d2 -= 2;  e1 *= e2;
        sum += dawson_constant4*(e1/d1 + 1/(d2*e1));
        d1 += 2;  d2 -= 2;  e1 *= e2;
        sum += dawson_constant5*(e1/d1 + 1/(d2*e1));
        result = M_1_SQRTPI*exp(-xp*xp)*sum;
    } else {
        float x2 = x*x;
        result = x*(1 - (2.0/3.0)*x2*(1 - .4*x2*(1 - (2.0/7.0)*x2)));
    }
    return result;
}
```

B.4 Volume Rendering Integral with Partial Pre-Integration

This listing is a function that will, given a pair input colors (with the alpha components set to the attenuation parameter τ), a ray segment length, and a Ψ table (which is ubiquitous), compute the volume rendering integral with linear interpolation for the luminance and attenuation. Section 5.2 introduces the partial pre-integration technique.

The function assumes that the Ψ table, which is stored in the `PsiTable` variable, is 1024 by 1024. The size can be changed by changing the definition of `PSI_TABLE_SIZE`.

Appendix A gives a *Mathematica* script that builds Ψ tables.

```
#define PSI_TABLE_SIZE        float2(1024,1024)
float4 integrateRay(in float4 colorBack, in float4 colorFront,
                    in float distance, in sampler2D PsiTable)
{
  float2 taudbackfront;
  taudbackfront.x = distance*colorBack.a;
  taudbackfront.y = distance*colorFront.a;
  float zeta = exp(-dot(taudbackfront, float2(0.5,0.5)));

  float2 gammabackfront = taudbackfront/(1+taudbackfront);
  float Psi = tex2D(PsiTable,
                    gammabackfront + float2(0.5,0.5)/PSI_TABLE_SIZE).w;

  float4 outColor;
  outColor.rgb = colorFront.rgb*(1-Psi) + colorBack.rgb*(Psi-zeta);
  outColor.a = 1 - zeta;

  return outColor;
}
```

B.5 Volume Rendering Integral with Linear Opacity and Luminance, Rough Approximation

This listing is a function that provides a rough but usually reasonable approximation to volume rendering with linearly interpolated opacity and luminance. Section 5.3.1 gives details of the approximation. We may use this function in conjunction with the fragment program given in Appendix B.2.

```
float4 IntegrateRay(in float4 BackColor, in float4 FrontColor,
                    in float Length)
{
    float dtau = -distance*LengthMultiply
                  *log(1-0.5*(colorBack.a+colorFront.a));
    float zeta = exp(-dtau);
    float alpha = 1 - zeta;
    float Psi = alpha/dtau;

    float4 color;
    color.rgb = colorFront.rgb*(1-Psi) + colorBack.rgb*(Psi-zeta);
    color.a = alpha;

    return color;
}
```

B.6 Volume Rendering Integral with Linear Opacity and Luminance, Close Approximation

This listing is a function that provides a close but reasonably fast approximation to volume rendering with linearly interpolated opacity and luminance. Section 5.3.2 gives details of the approximation. We may use this function in conjunction with the fragment program given in Appendix B.2.

```
float4 IntegrateRay(in float4 BackColor, in float4 FrontColor,
                    in float Length)
{
    float zeta = pow(1 - (  0.5*(colorBack.a+colorFront.a)
                          + (0.108165*(colorBack.a-colorFront.a)
                             *(colorBack.a-colorFront.a)) ),
                     distance*LengthMultiply);
    float alpha = 1 - zeta;
    float dtau2 = -distance*LengthMultiply
                  *log(1-(0.27*colorBack.a+0.73*colorFront.a));
    float zeta2 = exp(-dtau2);
    float Psi = (1-zeta2)/dtau2;

    float4 color;
    color.rgb = colorFront.rgb*(1-Psi) + colorBack.rgb*(Psi - zeta);
    color.a = alpha;

    return color;
}
```

References

[1] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*™. Addison Wesley, third edition, 2003. ISBN 0-201-77343-0.

[2] Howard Anton. *Elementary Linear Algebra*. Wiley, New York, eighth edition, 2000. ISBN 0-471-17055-0.

[3] Steven Bergner, Torsten Möller, Mark S. Drew, and Graham D. Finlayson. Interactive spectral volume rendering. In *Proceedings of IEEE Visualization 2002*, pages 101–108, October/November 2002.

[4] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (ACM SIGGRAPH 82)*, volume 16, pages 21–29, July 1982.

[5] Paul Bunyk, Arie Kaufman, and Cláudio Silva. Simple, fast, and robust ray casting of irregular grids. In *Proceedings of Dagstuhl '97*, pages 30–36, 1997.

[6] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–98, October 1994.

[7] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, volume 18, pages 103–108, July 1984.

[8] Edwin Catmull. A hidden-surface algorithm with antialiasing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 78)*, volume 12, pages 6–11, August 1978.

[9] Paolo Cignoni and Leila De Floriani. Power diagram depth sorting. In *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages

88–89, Montréal, Québec, Canada, 1998. School of Computer Science, McGill University.

[10] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann, August 1993.

[11] João Comba, James T. Klosowski, Nelson Max, Joseph S. B. Mitchell, Cláudio T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Eurographics '99)*, 18(3):369–376, April/June 1999.

[12] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993.

[13] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufmann. High-quality volume rendering using texture mapping hardware. In *1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 69–76, August 1998.

[14] Frank Dachille IX and Arie Kaufman. GI-Cube: An architecture for volumetric global illumination and rendering. In *2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–128, August 2000.

[15] Mark de Berg. *Lecture Notes in Computer Science*, volume 703, chapter Ray Shooting, Depth Orders and Hidden Surface Removal. Springer-Verlag, Berlin, 1993.

[16] Mark de Berg, Mark Overmars, and Otfried Schwarzkopf. Computing and verifying depth orders. *SIAM Journal on Computing*, 23(2):437–446, 1994.

[17] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000. ISBN 3-540-65620-0.

[18] Paul Joseph Diefenbach. *Pipeline Rendering: Interaction and Reasism Through Hardware-Based Multi-Pass Rendering*. PhD thesis, University of Pennsylvania, 1996.

[19] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of ACM SIGGRAPH 2000*, pages 19–28, July 2000.

[20] Robert Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Computer Graphics (ACM SIGGRAPH 88)*, volume 22, pages 65–74, August 1988.

[21] David Ebert and Penny Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, pages 195–201, October 2000.

[22] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In Stephen N. Spencer, editor, *2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 9–16, August 2001.

[23] A. M. Eskicioglu and P. S. Fisher. Image quality measures and their performance. *IEEE Transactions on Communications*, 43(12):2959–2965, December 1995.

[24] Ricardo Farias, Joseph S. B. Mitchell, and Cláudio T. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of the ACM/IEEE Volume Visualization and Graphics Symposium*, pages 91–99, 2000.

[25] Ricardo Farias and Cláudio T. Silva. Parallelizing the ZSWEEP algorithm for distributed-shared memory architectures. In *Proceedings of the International Volume Graphics Workshop, 2001*, 2001.

[26] James A. Ferwerda. Elements of early vision for computer graphics. *IEEE Computer Graphics and Applications*, 21(5):22–33, September/October 2001.

[27] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman Lectures on Physics*, volume I, chapter 35, Color Vision. Addison-Wesley, Reading, Massachusetts, 1977. ISBN 0-201-02010-6-H.

[28] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice*. Second Edition in C. Addison-Wesley, July 1997. ISBN 0-201-84840-6.

[29] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. In *Computer Graphics (ACM SIGGRAPH 83)*, volume 17, pages 65–72, July 1983.

[30] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (ACM SIGGRAPH 80)*, volume 14, pages 124–133, July 1980.

[31] Michael P. Garrity. Raytracing irregular volume data. In *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, volume 24, pages 35–40, November 1990.

[32] Christopher Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[33] Stefan Guthe, Stefan Röttger, Andreas Schrieber, Wolfgang Straßer, and Thomas Ertl. High-quality unstructured volume rendering on the PC platform. In *Proceedings of the SIGGRAPH/Eurographics Graphics Hardware Workshop 2002*, pages 119–125, 2002.

[34] Eric Haines. *Graphics Gems II*, chapter V.I, Fast Ray-Convex Polyhedron Intersection, pages 247–250. Academic Press, Orlando, FL, 1991. ISBN 0-12-064481-9.

[35] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989. ISBN 0-387-96774-5.

[36] Andrew J. Hanson. *Graphics Gems IV*, chapter II.6, Geometry for N-Dimensional Graphics, pages 149–170. Academic Press, San Francisco, CA, 1994. ISBN 0-12-336155-9.

[37] Mark J. Harris and Anselmo Lastra. Real-time cloud rendering. *Compute Graphics Forum (Eurographics 2001 Proceedings)*, 20(3):76–84, September 2001.

[38] Paul Heckbert. Ray tracing JELL-O® brand gelatin. In *Computer Graphics (ACM SIGGRAPH 87)*, pages 73–74, 1987.

[39] Norman Jouppi and Chun-Fa Chang. Z^3: An economical hardware technique for high-quality antialiasing and transparency. In *1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 85–93, August 1999.

[40] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. In *Computer Graphics (ACM SIGGRAPH 84)*, pages 165–173, July 1984.

[41] Alexander Keller. Instant radiosity. In *Proceedings of SIGGRAPH 1997*, pages 49–56, 1997.

[42] Rahul Khardekar and David Thompson. Rendering higher order finite element surfaces in hardware. In *Proceedings of GRAPHITE 2003*, pages 211–217, Febuary 2003.

[43] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, pages 79–86, October 1998.

[44] Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of IEEE Visualization 2003*, October 2003.

[45] Davis King, Craig M. Wittenbrink, and Hans J. Wolters. An architecture for interactive tetrahedral volume rendering. In *Volume Graphics 2001, Proceedings of the International Workshop on Volume Graphics 2001*, pages 163–180, June 2001.

[46] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of IEEE Visualization 2001*, pages 255–262, October 2001.

[47] 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, July/September 2002.

[48] Joe Kniss, Simon Premože, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, pages 109–116, October/November 2002.

[49] Joe Kniss, Simon Premože, Milan Ikits, Aaron Lefohn, and Charles Hansen. Closed-form approximations to the volume rendering integral with gaussian transfer functions. Technical Report UUCS-03-013, School of Computing, University of Utah, July 2003.

[50] Joe Kniss, Simon Premože, Milan Ikits, Aaron Lefohn, Charles Hansen, and Emil Praun. Gaussian transfer functions for multi-field volume visualization. In *Proceedings of IEEE Visualization 2003*, pages 497–504, October 2003.

[51] 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, April/June 2003.

[52] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In *Proceedings of IEEE Visualization 2001*, pages 215–222, October 2001.

[53] Kevin Kreeger and Arie Kaufman. PAVLOV: A programmable architecture for volume processing. In *1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 77–86, August 1998.

[54] Shankar Krishnan, Cláudio T. Silva, and Bin Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering. In *Proceedings of Data Visualization (Eurographics/IEEE Symposium on Visualization)*, pages 233–242, 2001.

[55] Philippe Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.

[56] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Proceedings of ACM SIGGRAPH 94*, volume 28, pages 451–457, July 1994.

[57] C. Charles Law, Amy Henderson, and James Ahrens. An application architecture for large data visualization: A case study. In *Proceedings of the 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 125–128, October 2001.

[58] Joshua Leven, Jason Corso, Jonathan Cohen, and Subodh Kumar. Interactive visualization of unstructured grids using hierarchical 3D textures. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 37–44, October 2002.

[59] Erick Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, pages 149–157, August 2001.

[60] Tom Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[61] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming grapihcs hardware in a C-like language. *ACM Transactions on Graphics (ACM SIGGRAPH 2003)*, 22(3):896–907, July 2003.

[62] Nelson Max. Light diffusion through clouds and haze. *Computer Vision, Graphics, and Image Processing*, 33:280–292, 1986.

[63] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[64] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, volume 24, pages 27–33, December 1990.

[65] A. Mayache, T. Eude, and H. Cherifi. A comparison of image quality models and metrics based on human visual sensitivity. In *IEEE International Conference on Image Processing*, volume 3, pages 409–413, October 1998.

[66] Michael D. McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical Report CS-2000-14, University of Waterloo, August 2000.

[67] M. Meißner, U. Kanus, and W. Straßer. VIZARD II, a PCI-card for real-time volume rendering. In *1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 61–67, August 1998.

[68] Michael Meißner, Ulrich Hoffman, and Wolfgang Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of IEEE Visualization 1999*, pages 207–214, October 1999.

[69] Jurriaan D. Mulder, Frans C.A. Groen, and Jarke J. van Wijk. Pixel masks for screen-door transparency. In *Proceedings of IEEE Visualization '98*, pages 351–358, 1998.

[70] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM National Conference*, volume 1, pages 443–450, 1972.

[71] Kevin Lawrence Novins. *Towards Accurate and Efficient Volume Rendering*. PhD thesis, Cornell University, January 1994.

[72] Michael S. Paterson and F. Frances Yao. Binary partitions with applications to hidden-surface removal and solid modelling. In *Symposium on Computational Geometry*, pages 23–32, 1989.

[73] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time ray-casting system. In *Proceedings of ACM SIGGRAPH 99*, pages 251–260, August 1999.

[74] Thomas Porter and Tom Duff. Compositing digital images. In *Computer Graphics (ACM SIGGRAPH 84)*, volume 18, pages 253–259, July 1984.

[75] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*, pages 221–227. Cambridge, second edition, 2002. ISBN 0-521-750332-4.

[76] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*, pages 264–265. Cambridge, second edition, 2002. ISBN 0-521-750332-4.

[77] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetko, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, pages 159–170, August 2001.

[78] Harvey Ray and Deborah Silver. The Race II engine for real-time volume rendering. In *2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 129–136, August 2000.

[79] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In Stephen N. Spencer, editor, *2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–118, August 2000.

[80] Stefan Röttger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, pages 23–28, October 2002.

[81] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 109–116, October 2000.

[82] George B. Rybicki. Dawson's integral and the sampling theorem. *Computers in Physics*, 3(2):85–87, March/April 1989.

[83] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. In *Computer Graphics (ACM SIGGRAPH 88)*, volume 22, pages 51–58, August 1988.

[84] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit*. Pearson Education, Inc, third edition, 2002. ISBN 1-930934-07-6.

[85] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (ACM SIGRAPH 92)*, 26(2):249–252, July 1992.

[86] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (Proceedings San Diego Workshop on Volume Visualization)*, volume 24, pages 63–70, December 1990.

[87] Cláudio T. Silva. *Parallel Volume Rendering of Irregular Grids*. PhD thesis, State University of New York at Stony Brook, November 1996.

[88] Cláudio T. Silva and Joseph S. B. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):142–157, April-June 1997.

[89] Cláudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman. Fast rendering of irregular grids. In *Proceedings of the ACM/IEEE Volume Visualization Symposium '96*, pages 15–22, 1996.

[90] Cláudio T. Silva, Joseph S. B. Mitchell, and Peter L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization*, pages 87–94, 1998.

[91] John Snyder and Jed Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. In *Proceedings of ACM SIGGRAPH 98*, pages 219–230, July 1998.

[92] Clifford Stein, Barry Becker, and Nelson Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 83–89, October 1994.

[93] T. Totsuka and M. Levoy. Frequency domain volume rendering. In *Proceedings of ACM SIGGRAPH 93*, pages 271–278, July 1993.

[94] Fan-Yin Tzeng, Eric B. Lum, and Kwan-Liu Ma. A novel interface for higher-dimensional classification of volume data. In *Proceedings of IEEE Visualization 2003*, pages 505–512, October 2003.

[95] A. van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *Volume Visualization Symposium Proceedings*, pages 23–30, October 1996.

[96] Zhou Wang and Alan C. Bovik. A universal image quality index. *IEEE Signal Processing Letters*, 9(3):81–84, March 2002.

[97] Manfred Weiler and Thomas Ertl. Hardware-software-balanced resampling for the interactive visualization of unstructured grids. In *Proceedings of IEEE Visualization 2001*, pages 199–206, October 2001.

[98] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-based view-independent cell projection. In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, pages 13–22, October 2002.

[99] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based ray casting for tetrahedral meshes. In *Procceedings of IEEE Visualization 2003*, pages 333–340, October 2003.

[100] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, April/June 2003.

[101] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of ACM SIGGRAPH 1998*, pages 169–177, July 1998.

[102] Jane Wilhelms and Allen van Gelder. A coherent projection approach for direct volume rendering. In *Computer Graphics (ACM SIGGRAPH 91)*, volume 25, pages 275–284, July 1991.

[103] P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[104] Peter Williams and Nelson Max. A volume density optical model. In *Computer Graphics (Proceedings of the 1992 Workshop on Volume Visualization)*, pages 61–68, October 1992.

[105] Peter L. Williams, Nelson L. Max, and Clifford M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), March 1998.

[106] Orion Wilson, Allen van Gelder, and Jane Wilhelms. Direct volume rendering via 3D textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, June 1994.

[107] Craig M. Wittenbrink. R-buffer: A pointerless A-buffer hardware architecture. In *2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 73–80, August 2001.

[108] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, fifth edition, 2003. ISBN 1-57955-022-3.

[109] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley, third edition, 1999. ISBM 0-201-60458-2.

[110] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 7–12, October 2002.

[111] Roni Yagel, David M. Reed, Asish Law, Po-Wen Shih, and Naeem Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proceedings of the 1996 Symposium on Volume Visualization*, pages 55–62, 1996.

[112] Caixia Zhang and Roger Crawfis. Volumetric shadows using splatting. In *Proceedings of IEEE Visualization 2002*, pages 85–92, October/November 2002.

[113] 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, April/June 2003.

Index

Ψ, 86

Ψ clipping, 89

α, 18, 95

τ, 13

γ, 91

ζ, 86

A-buffer, 48

adaptive transfer func. sampling, 73–82, 84

alpha (α), 18, 95

attenuation

- coefficient, 13
- versus opacity, 93

back to front rendering, 21

balanced cell projection, 70–73

basis graph, 38

binary space partitioning trees, 45

blending, 18

boundary faces, 32

BSP-XMPVO, 47

BSP trees, 45

cell-ray intersections, 29

cells, 2, 3

- connectivity graph, 33, 46
- linear, 3
- nonlinear, 3, 67

cell projection, 5, 35

- adaptive trans. func. sampling, 73–82, 84
- balanced, 70–73
- GATOR, 38–39
- projected tetrahedra, 36–37
- view independent, 39–43, 66–70

cell sorting, *see* visibility sorting

classification, 63

color computations, 29

color spectrum, 15

complementary error function, 58

compositing, 18

conforming mesh, 32

connected, 46

connectivity, 4

connectivity graph, 33, 46

control point, 75

convex, 36, 46

convex hull, 46

DAG, 47
Dawson's integral, 59
Delaunay triangulation, 47
depth sorting, *see* visibility sorting
directed acyclic graph, 47
direct volume visualization, 2

erf, 27
erfc, 58
erfi, 28
error function, 27
    complementary, 58
    imaginary, 28
external faces, 32

faces
    external, 32
    internal, 32
fragment, 8
fragment processing, 8, 52
frame buffer, 8
front to back rendering, 22

gamma (γ), 91
ganglion cell, 115
GATOR, 38–39
Gaussian, 61
geometric processing, 8, 30
geometry, 3
glow, 17
gradient, 42
graphics hardware
    volume rendering, 5–8
graphics pipeline, 6
grid, *see* mesh

hole, 46

image-order rendering, 31
image blending, 18
image compositing, 18
imaginary error function, 28
intensity, 14
internal faces, 32

linear cells, *see* cells, linear
luminance, 14

Mach bands, 56, 115
mesh, 3
    cell, 3
    conforming, 32
    connected, 46
    connectivity, 4
    convex, 46
    geometry, 3
    hole, 46
    rectilinear, 4

structured, 4
topology, 3
uniform, 4
unstructured, 3, 5
vertex, 3
MPVO, 46
MPVOC, 48
MPVONC, 47
multiple scattering, 12

nonlinear cells, *see* cells, nonlinear
normal distribution, *see* Gaussian

object-aligned slicing, 51
object-order rendering, 35
opacity, 18, 93
versus attenuation, 93
optical model, 10
order independent, 23
ostrich algorithm, 45
over operator, 20

partial pre-integration, 88
pixel, 8
Porter and Duff, 18
power distance, 48
pre-integration, 62
primitive, 8
projected tetrahedra, 36–37
Psi (Ψ), 86
Psi clipping, 89

R-buffer, 48
rasterization, 35
rasterize, 8, 69
ray casting, 5, 31
ray tracing, 31
rectilinear mesh, 4
RGB color space, 16
Riemann sum, 25, 53

scattering, 11
screen door transparency, 48
shadowing, 12
shear-warp factorization, 50
simple, 36
simplex, 36
structured mesh, 4
sweep, 34
line, 34
plane, 34, 47

tau (τ), 13
topology, 3
transfer function, 60, 62

under operator, 20
uniform mesh, 4
unstructured mesh, 5

vertex, 3, 8

view-aligned slicing, 52
view independent cell projection, 39–43, 66–70
visibility cycle, 45
visibility sorting, 23, 43–49
    BSP-XMPVO, 47
    BSP Trees, 45
    MPVO, 46
    MPVOC, 48
    MPVONC, 47
    XMPVO, 47
volume model, 11–13
volume rendering, 2
    hardware accelerated, *see* graphics hardware, volume rendering
volume rendering integral, 10–28
    closed forms, 22–28
    derivation, 11–15
    equation for, 14–15
    model, 11–13
    properties, 15–22
voxel, 2, 4

wavelength, 15

XMPVO, 47

Z^3, 48
z-buffer algorithm, 48
zeta (ζ), 86
ZSWEEP, 49