VOLUME VISUALIZATION USING ADVANCED GRAPHICS HARDWARE SHADERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

Daqing Xue, M.S.

*****

The Ohio State University
2008

Dissertation Committee:

Professor Roger Crawfis, Adviser

Professor Raghu Machiraju

Professor Han-Wei Shen

Approved by

Adviser
Computer Science and Engineering
Graduate Program

ABSTRACT

Graphics hardware based volume visualization techniques have been the active research topic over the last decade. With the more powerful computation ability, the availability of large texture memory, and the high programmability, modern graphics hardware has been playing a more and more important role in volume visualization.

In the first part of the thesis, we focus on the graphics hardware acceleration techniques. Particularly, we develop a fast X-Ray volume rendering technique using point-convolution. An X-ray image is generated by convolving the voxel projection in the rendering buffer with a reconstruction kernel. Our technique allows users to interactively view large datasets at their original resolutions on standard PC hardware. Later, an acceleration technique for slice based volume rendering (SBVR) is examined. By means of the early z-culling feature from the modern graphics hardware, we can properly set up the z-buffer from isosurfaces to gain significant improvement in rendering speed for SBVR.

The high programmability of the graphics processing unit (GPU) incurs a great deal of research work on exploring this advanced graphics hardware feature. In the second part of the thesis, we first revisit the texture splat for flow visualization. We develop a texture splat vertex shader to achieve fast animated flow visualization.

Furthermore, we develop a new rendering shader of the implicit flow. By careful tracking and encoding of the advection parameters into a three-dimensional texture, we achieve high appearance control and flow representation in real time rendering. Finally, we present an *indirect shader* synthesizer to combine different shader rendering effects to create a highly informative image to visualize the investigating data. One or more different shaders are associated with the voxels or geometries. The shader is resolved at run time to be selected for rendering. Our indirect shader synthesizer provides a novel method to control the appearance of the rendering over multi-shaders.

To my daughters and wife with love

ACKNOWLEDGMENTS

I would like first to thank my advisor, Roger Crawfis, for taking me as his student and allowing me to pursue my research interests, and for his guidance, encouragement and friendship during the course of this work.

I would like to thank my committee members Han-Wei Shen and Raghu Machiraju for helping make this thesis happen and their valuable suggestions. I must thank professor Rephael Wenger to help me understand the high-dimensional iso-contouring.

I am very thankful for the many people at graphics lab that have offered help and insightful discussions. In particular, I thank Udeepta Bordoloi, Antonio Garcia, Jinzhu Gao, Samrat Goswami, Yuan Hong, Guangfeng Ji, Ming Jiang, Jinho Lee, Guo-Shi Li, Liya Li, Naeem Shareef, Jian Sun, Chaoli Wang, Lining Yang, Caixia Zhang, and Wulue Zhao.

I wish to thank all my co-workers at Siemens Medical that have offered help and friendly work environment. Special thanks to Min Xie for keeping me the top-line graphics hardware on doing research in this thesis.

Finally I thank to my wife, Fang Yuan, for her love and support, and my two daughters, Yinbing and Wenchan, who bring me a great deal of fun and happiness to go through the tough process of finishing this thesis.

VITA

October 3, 1971 ……... Born - Wuhu, China

1999 ……. M.S. Computer Science,
Nankai University, China

1999-2001 ……... Graduate Research Associate,
Department of Biomedical Engineering,
Cleveland Clinic Foundation,
The Ohio State University

2001-2005 ……... Graduate Research and Administrative Associate,
Department of Computer Science and Engineering,
The Ohio State University

2005 - present …….. Sr. Scientist,
Siemens Medical Solutions, USA

PUBLICATIONS

Research publication related to this work:

1. Daqing Xue, Caixia Zhang, Roger Crawfis, "iSBVR: Isosurface-aided Hardware Acceleration Techniques for 3D Slice-Based Volume Rendering," *International Workshop on Volume Graphics 2005*.

2. Daqing Xue, Caixia Zhang, Roger Crawfis, "Rendering Implicit Flow Volumes," *Proceedings of IEEE Visualization 2004*, pages 99-106, Austin, TX, October 2004.

3. Praveen Baniramka, Caixia Zhang, Daqing Xue, Roger Crawfis, Rephael Wenger, "Volume Interval Segmentation and Rendering," *IEEE/SIGGRAPH Symposium on Volume Visualization 2004*, pages 55-62, Austin, TX, 2004. (Best Paper Award)

4. Roger Crawfis, Daqing Xue, Caixia Zhang, "Volume Rendering Using Splatting," *Visualization Handbook*, eds. Charles Hansen, Christopher Johnson, pages 175-188, Academic Press, 2004.

5. Daqing Xue and Roger Crawfis, "Fast Dynamic Flow Volume Rendering Using Textured Splats on Modern Graphics Hardware," *Proceedings of SPIE EI 2004*, pages 133-140, San Jose, CA, 2004.

6. Daqing Xue and Roger Crawfis, "Efficient Splatting Using Modern Graphics Hardware," *Journal of Graphics Tools*, Vol. 8, No. 3, pages 1-21, 2003.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

VOLUME VISUALIZATION FUNDAMENTALS

1.1 Volume Data Representation

Unlike polygons defining an object model in surface-based computer graphics, volume data is typically defined as a set, $\boldsymbol{V}$, of samples. This is a function representing the information $\boldsymbol{v}$ of the data at the position (x, y, z) in three-dimensional space $\Re^3$. This information can be a *scalar* (such as density in a CT scan), a *vector* (such as velocity in a flow), or a *tensor* (such as density, energy, and temperature in *computational fluid dynamics*). In addition, if the volume data is time-varying, the volume is then defined in 4D space with samples $\boldsymbol{v}$ (x, y, z, t).

In many scientific applications, the volume is sampled on a *rectilinear grid*, of which each volume element is called a *voxel*. For such a rectilinear grid, the voxel set, $\boldsymbol{V}$, can be redefined as an array of data values $V(i, j, k)$, which are sampled only at the grid points indexed by (i, j, k) in volume space, and the dataset is thus depicted by a 3D array. Figure 1.1 shows a voxelized object and a *cell* with eight neighboring voxels on a rectilinear grid. If all the voxels are spaced equally in all dimensions, the dataset is called

regular or *isotropic*. Other types of datasets include *curvilinear* and *unstructured* based on underlying computational grids for sampling. Curvilinear grids are non-linear transformations from rectilinear grids while preserving the grid topology. Curvilinear grids are useful to describe the simulations with variable resolution. An unstructured volume consists of polyhedral cells whose connectivity has to be specified explicitly. These cells can be tetrahedra, pyramids, hexahedra, or other shapes. Many datasets for *finite element simulation* (FEM) are provided in unstructured grids. Figure 1.2 demonstrates the different computational grids for volume datasets.

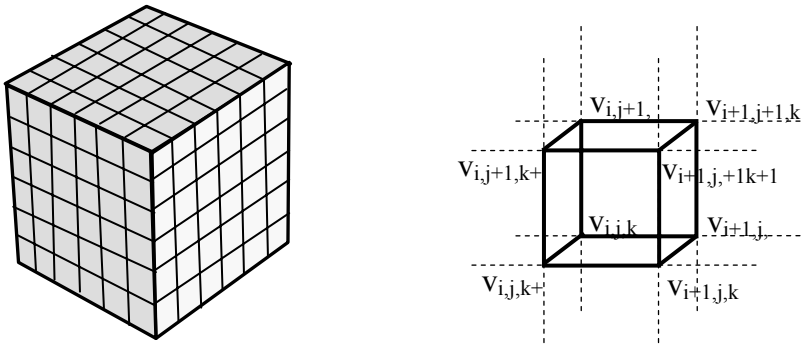


Figure 1.1: A voxlized object (left) and a cell (right) with eight neighboring voxels on a rectilinear grid.

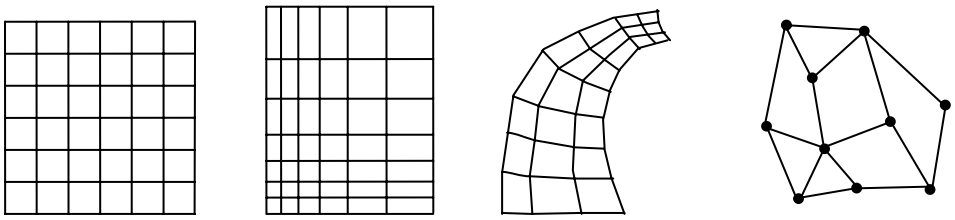


Figure 1.2: The 2D computational grids: regular, rectilinear, curvilinear, and unstructured.

1.2 Volume Reconstruction

A scalar field volume can be interpreted as a scalar function, $f : \Re^3 \to \Re$, and the volumetric dataset, *V*, is a collection of samples from the continuous function *f* at the grid points. The continuum of the volume can be reconstructed by convolving these discrete samples with a *reconstruction filter*. According to sampling theory, this reconstruction can be lossless, provided that the original function *f* is band-limited and the sampling frequency is at least twice the highest frequency (*Nyquist frequency*) in the function *f*. An ideal reconstruction filter must be employed to achieve such a lossless reconstruction, such as the "*sinc*" function (in equation 1.1).

$$\text{sinc(x)} = \frac{\sin(\pi x)}{\pi x} \tag{1.1}$$

The graph of the *sinc* function is shown in Figure 1.3a. Note this function has infinite spatial extent. This implies that all samples must be considered for the reconstruction of any point in the volume space. Thus a lossless reconstruction is

computationally expensive. A truncated *sinc* or other reconstruction filters such as a box, triangle, and truncated *Gaussian* filters can be used instead. This can lead to artifacts in the reconstructed volume. Figure 1.3 shows the graphs of these 1D filters. On the other hand, such a continuous function *f* can be approximated by a piecewise linear function that defines on the underlying grid. In this case, a *tri-linear interpolation* can be used to compute the function values on the non-grid points.

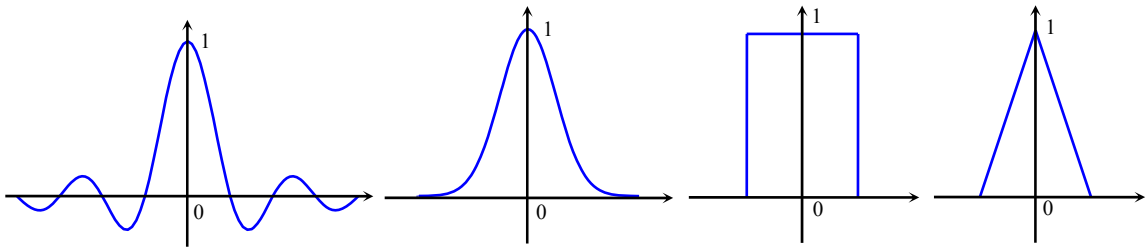


Figure 1.3: 1D reconstruction filters. From left to right: sinc, Gaussian, box, and triangle.

1.3 Indirect Volume Rendering

For a scalar function, $f(x,y,z)$, that defines a volume in $\Re^3$, $f(x,y,z)=c$ defines an isosurface that describes the boundary with data value c. Extracting and rendering such surface primitives provides a way to indirectly render the volume. The *Marching-cubes* algorithm developed by Lorensen and Cline [LC87] creates an elegant approximation to an isosurface with a triangle mesh. In the marching-cubes algorithm, cubes containing

isosurface manifolds are segmented according to the iso-value *c* to generate surface patches. Finally these patches are triangulated and form a triangle mesh that approximates the underlying isosurface. Figure 1.4 shows the 15 reduced segmented cube cases used in [LC87]. Figure 1.5 demonstrates isosurface images from some sample datasets.

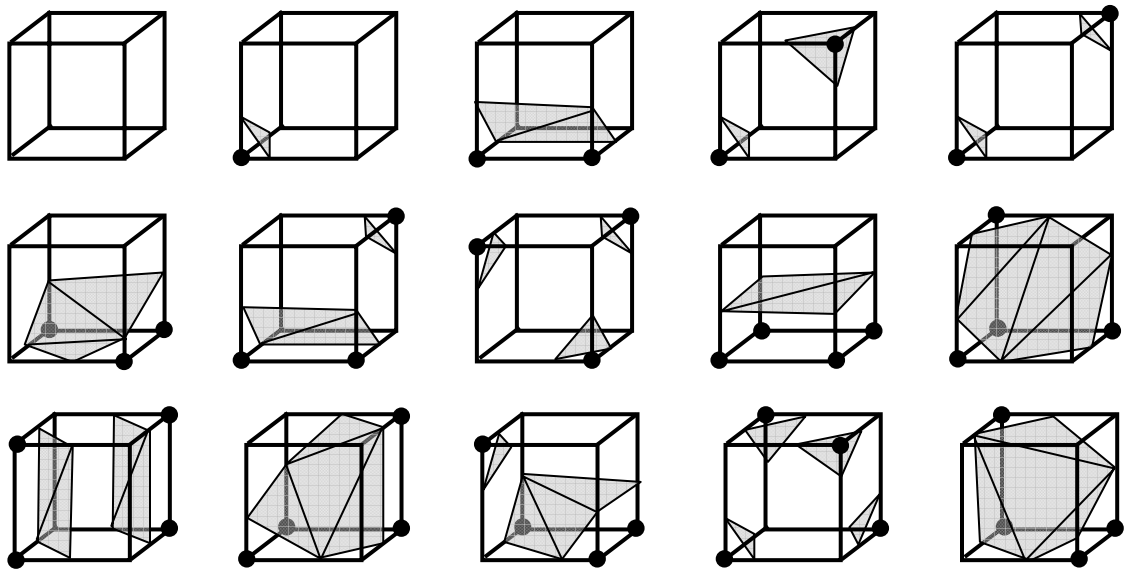


Figure 1.4: The 15 reduced triangulated cube cases [LC87].

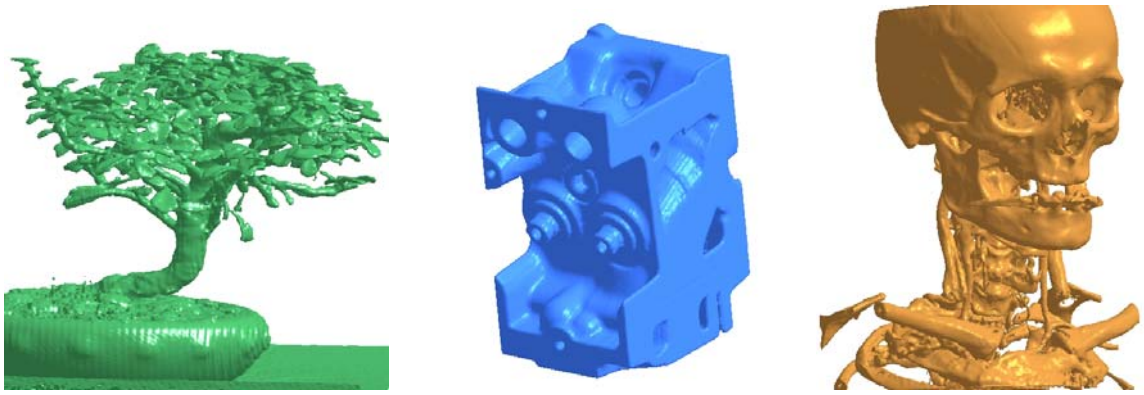


Figure 1.5: Isosurfaces for CT datasets of bonsai, engine, and head.

1.4 Direct Volume Rendering

Direct volume rendering (DVR) generates an image of the volume by integrating the light effect along viewing rays that traverse the entire volume based on a given optical model. No intermediate geometric primitives are extracted to represent a volume object. The rendering time complexity is $o(n^3)$, which makes real-time volume rendering difficult. This has lead to many research efforts on accelerated direct volume rendering [Lev90, DH92, YS93, FS97, ASK92, SA95].

1.4.1 The Volume Rendering Integral

In the volume rendering integral, optical models describe the light interaction (*absorption*, *emission*, and/or *scattering*) amongst particles in the volume and scattered forwards to the eye-point. Max [Max95] examined five optical models: *absorption-only*, *emission-only*, *absorption-plus-emission*, *scattering/shadow*, and *multi-scattering*, in his survey paper.

The most commonly-used models: absorption-only and absorption-plus-emission, are briefly described in the following two sub-sections. More details for the other models can be found in [Max95].

1.4.1.1 X-Ray and the Absorption-only Model

For the absorption-only model, the function, *f(x,y,z)*, can be mapped to an extinction coefficient, τ, that controls the rate that light is occluded. Max [Max95] derives the mathematical formula for this simple model. The differential change of the light intensity, *I*, along a ray can be written as:

$$\frac{dI}{ds} = -\tau(s)I(s) \tag{1.2}$$

where *s* is a length parameter along a ray in the direction of the light flow. This equation states that the change in the intensity (*dI/ds*) decreases, hence the negative multiplier, proportionally to the incoming intensity as determined by the extinction coefficient. The analytical solution for this formula is given by:

$$I(s) = I_0 e^{-\int_0^s \tau(t)dt} \tag{1.3}$$

where I_0 is the intensity at $s=0$, where the ray enters the volume. This indicates the attenuation of the high energy source or backlight as it propagates from the background, s=0, towards the eye. If τ is zero along the ray, then no attenuation occurs and the intensity at the pixel is I_0. If τ is a constant along this ray, then the attenuation is given by:

$$I(s) = I_0 e^{-\int_0^s \tau dt} = I_0 e^{-\tau s} \tag{1.4}$$

Using a Taylor's series expansion for the exponential and simplifying for the case where τs is small leads to the familiar compositing operator, *over*, from Porter and Duff [PD84]:

$$\begin{aligned} I_0 e^{-\tau s} &= I_0 \left(1 - \tau s + \frac{(\tau s)^2}{2!} - \frac{(\tau s)^3}{3!} + \cdots \right) \\ &\approx I_0 (1 - \tau s) \end{aligned} \tag{1.5}$$

Here, τs represents the opacity, α, expressed as a function of the traversing ray length. As Max [Max95] and Wilhelms and van Gelder [WvG91] point out, this relationship of increased opacity for longer ray integration segments is crucial when considering different sampling resolutions of the volume. This equation implies that for volume rendering using relatively low opacity values, the simple *over* operator is probably sufficient. Note low opacity results from small values of s and/or τ in equation 1.5.

When performing the integration in equation 1.3 for all rays propagating a volume, the integral result is an X-Ray image. X-ray imaging is one of most popular visualization techniques used in clinical diagnosis and in material defect detection in industrial applications. The advantage of X-ray volume rendering is that no sorting of the voxels is needed. This is particularly beneficial for object-order rendering techniques like *splatting* [Wes90].

1.4.1.2 The Absorption-plus-emission Model

The absorption-plus-emission model is introduced by Sabella [Seb88] and elaborated by

Max [Max95]. The volume rendering integral for this model is:

$$I(s) = I_0 e^{-\int_0^s \tau(r)dr} + \int_0^s g(t) e^{-\int_t^s \tau(r)dr} dt \tag{1.6}$$

Here, the background light is attenuated, as in the absorption-only model, but new energy is scattered towards the eye along the ray according to the *glow function, g(t)*. This newly added energy is then attenuated based on the length of material that still exists between it and the eye. Equation 1.6 can be solved by numerical integration. A Riemann sum is used to approximate the integral with $\sum_{i=0}^{n} f(i\Delta t)\Delta t$, where the integral length is discretized into $n=s/\Delta t$ intervals of width Δt. Thus the solution to equation 1.6 can be approximated as:

$$\begin{aligned} I(s) &\approx I_0 \exp\left(-\sum_{i=1}^{n} \tau(i\Delta t)\Delta t\right) + \sum_{i=1}^{n} g(i\Delta t) \exp\left(-\sum_{j=i+1}^{n} \tau(j\Delta t)\Delta t\right)\Delta t \\ &= I_0 \prod_{i=1}^{n} \exp\left(-\tau(i\Delta t)\Delta t\right) + \sum_{i=1}^{n} g(i\Delta t)\Delta t \prod_{j=i+1}^{n} \exp\left(-\tau(j\Delta t)\Delta t\right) \end{aligned} \tag{1.7}$$

Let $g_i = g(i\Delta t)\Delta t$ and $\exp(-\tau(i\Delta t)\Delta t)$ be $1-\alpha_i$, where α_i is the material opacity of the i^{th} interval along the ray. Then equation 1.7 can be rewritten as:

$$\begin{aligned} I(s) &= I_0 \prod_{i=1}^{n} (1-\alpha_i) + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} (1-\alpha_j) \\ &= (\cdots((I_0(1-\alpha_1)+g_1)(1-\alpha_2)+g_2)+\cdots)(1-\alpha_{n-1})+g_n \end{aligned} \tag{1.8}$$

Equation 1.8 indicates that volume integral on this optical model can be performed by means of an *over* operator in a back-to-front order or a front-to-back order (see [Max95]).

1.4.2 Direct Volume Rendering Techniques

The direct volume rendering methods can be divided into *image-order* and *object-order* techniques. Image-order techniques compute the volume integral along the ray for each pixel of the resulting image. Rays shoot through pixels on the image plane into the volumetric dataset and Riemann summation in equation 1.8 is taken for the volume integrals along rays. This is a typical way to perform DVR termed *Ray-casting* [Seb88, KvH84]. On the other hand, object-order techniques project each voxel of the volume onto the image plane and distribute its contribution to the final image. Volume samples (voxels) are convolved with a pre-integrated reconstruction filter (or called *footprints*) and composited into the frame buffer. Westover [Wes89, Wes90] introduced such a *splatting* technique to implement an object-order algorithm. Another object-order based algorithm is shear-warp introduced by Lacroute and Levoy [LL94], in which samples in a dataset are traversed in an object-order and Riemann summation in equation 1.8 is taken along rays that shoot through the sheared volume. The integral result is then warped to generate the final image.

Another category for direct volume rendering techniques is *frequency-domain volume rendering* (FVR) developed by Malzbender [Mal93] and Totusk and Levoy [TL93]. The entire volume is transformed into frequency domain in a pre-processing step. A view-dependent projection image of the input volume in spatial space is computed by extracting a slice in frequency domain and transforming back to spatial domain with the inverse 2D Fourier transformation. The main advantage of FVR techniques is their runtime computational complexity of $o(n^2)$ for inverse 2D Fourier transformations.

However, frequency domain techniques are limited to parallel projection and X-ray-like compositing.

1.5 Classification

In direct volume rendering, *classification* is a process that maps the physical properties of the volume, such as density in a CT-scanned dataset, to optical properties like *emission* (color) and *absorption* (opacity). This mapping is usually performed via a *transfer function*. A transfer function is defined as:

$$T(\vec{\omega}) : \wp \rightarrow \Theta \tag{1.9}$$

where $\wp$ denotes a physical property space, and Θ denotes an optical property space in which each element is a 4-tuple as RGBA. In the simplest case, $\vec{\omega}$ is only the intensity of the volume at a sampling point. Other physical properties, such as the magnitude of the gradient, can be included to make a multi-dimensional transfer function [KKH01]. When different transfer functions are applied to the same volume, different interior structures of the volume can be visualized (see Figure 1.6). Depending on the stage in which classification is applied, we can distinguish between *pre-classification* and *post-classification*. In pre-classification, the voxels are classified and associated with a RGBA color (optical property) such that the color is interpolated in the following volume rendering integral. On the other hand, voxel values (physical property) are interpolated first and then the per-sample interpolated values are mapped to their optical property (RGBA) in a post-classification. These two classifications lead to different visual effects with blurred features in the former and sharp features in the latter.

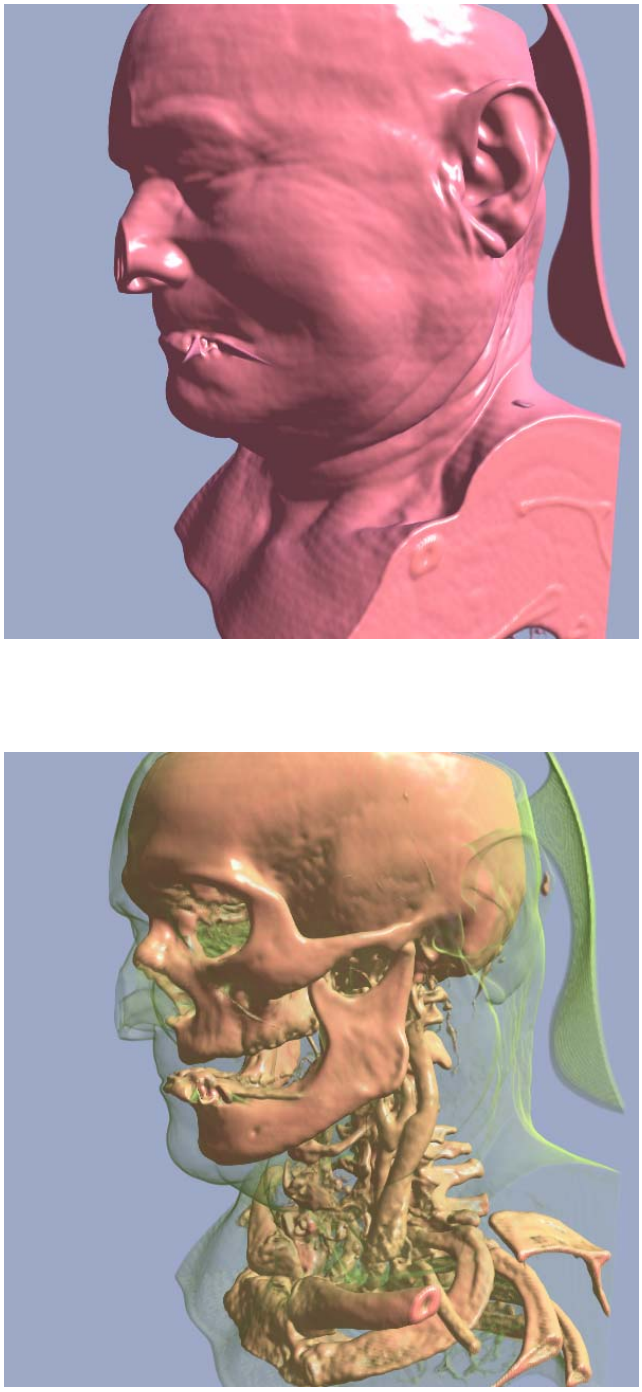


Figure 1.6: Two different transfer functions for a CT head dataset.

PART I

GRAPHICS HARDWARE ACCELERATION TECHNIQUES

CHAPTER 2

FAST X-RAY VOLUME RENDERING

2.1 Introduction

X-ray imaging is one of most popular visualization techniques used in clinical diagnosis and in material defect detection in industry. X-ray images can be generated from volumetric data using an X-ray (absorption only) optical model [Max95, XC03]. The advantage of X-ray volume rendering is that sorting of the voxels is not needed for monochrome color image due to the X-Ray model [Cra96]. This algorithm is especially beneficial for object-order rendering techniques such as *splatting* [Wes89], because it is simply to maintain a sorted list of data values along with their corresponding coordinate positions. Rendering these X-ray like images is equivalent to solving the integrals along the view direction, since the exponential function needs to be computed only once per output pixel. Malzbender [Mal93] and Totsuka and Levoy [TL93] have shown how to use the Fourier projection slice theorem and fast Fourier transforms to compute these integrals very rapidly.

Computational time complexity is a crucial factor to determine the frame rate and evaluate the performance of a specified volume rendering technique. In general, the time

complexity for object-order based volume rendering algorithms is $O(N^3)$ for a volume with size of N^3; for the image-order algorithm such as *ray-casting*, the time complexity is determined by the image resolution and the number of samples along the rays. The basic operation unit measured in time complexity can be the tri-linear interpolation sampling the volume for *ray-casting* or the sample point scattering for *splatting*. Per-fragment operations like lighting could be included in the operation unit. Such unit computation time varies greatly from the software-based rendering to the hardware-accelerated rendering. The frame rate can reach 5-10 FPS for volumes with sizes of 256^3 on the latest graphics hardware [XZC05, KW03] if no other acceleration techniques are used. The main drawback of the hardware techniques is that the volume size is limited by the expensive texture memory on the graphics hardware. The maximum volume size cannot easily exceed 512^3 for the latest NVIDIA GeForce 6800 [NVIDIAa] hardware due to its limited amount of texture memory. The same problems also arise for special-purpose graphics hardware like the Volume Pro [PHK*99]. By transforming the input volume into frequency domain, Malzbender [Mal93] and Totusk and Levoy [TL93] developed Fourier volume rendering (FVR) which reduces the time complexity to $O(N^2 logN)$. More recently, a Quasi-Monte Carlo volume rendering technique [CS03] was introduced with a time complexity of $O(N^2)$.

In this chapter, we provide a pragmatic and efficient X-ray image visualization technique to allow users to interactively view such large datasets at their original resolutions on standard PC hardware.

2.2 X-ray Convolution

Considering the input volume, V, as a finite number of samples, $f(\mathbf{x}_{i,j,k})$, sampling from a continuous 3D density function, $f(\mathbf{x})$, at regular grid points in $\Re^3$, the continuum of the volume can be reconstructed with an appropriate kernel, $h(\mathbf{x})$, as in equation 2.1:

$$f(\mathbf{x}) = \sum_{\mathbf{x}_{i,j,k} \in V} f(\mathbf{x}_{i,j,k}) h(\mathbf{x} - \mathbf{x}_{i,j,k}) \tag{2.1}$$

Where $\mathbf{x} \in (x, y, z)$ is in volume space.

In a standard X-ray volume rendering model [Max95], a ray from the eye to each pixel in the image is cast through the volume and the volume integral along the ray l is calculated as the density absorption. The volume integral for an arbitrary view direction w is described in equation 2.2:

$$I = \int^{w} f(\mathbf{x}) d\boldsymbol{l} = \int^{w} \sum_{\mathbf{x}_{i,j,k} \in V} f(\mathbf{x}_{i,j,k}) h(\mathbf{x} - \mathbf{x}_{i,j,k}) d\boldsymbol{l} \tag{2.2}$$

Samples in eye space can be represented as $\widetilde{f}(\mathbf{x}_{s,t,r}) = f(\mathbf{M}\mathbf{x}_{i,j,k})$, where $\mathbf{M}$ is the transformation matrix from volume space to eye space. Thus, for a parallel projection, the volume integral at the position (u, v) on the image plane can be rewritten from equation 2.2 as follows:

$$\begin{aligned} \mathrm{I}(\mathrm{u}, \mathrm{v}) &= \int^{w} \sum_{\mathbf{x}_{s,t,r} \in V} \widetilde{f}(\mathbf{x}_{s,t,r}) h(\mathbf{x} - \mathbf{x}_{s,t,r}) dw \\ &= \sum_{\mathbf{x}_{s,t,r} \in V} \widetilde{f}(\mathbf{x}_{s,t,r}) \int^{w} h(\mathbf{x} - \mathbf{x}_{s,t,r}) dw \end{aligned} \tag{2.3}$$

Where $\mathbf{x} \in (u, v, w)$ is in eye space.

The 2D footprint, $footprint_h(u,v)$ of a 3D reconstruction kernel, $h(u,v,w)$ is given by [Wes89]:

$$footprint_h(u,v) = \int_{-\infty}^{\infty} h(u,v,w)dw \tag{2.4}$$

If the kernel is restricted to be rotationally symmetric, equation 2.3 can be reduced to:

$$I(u,v) = \sum_{(s,t,r)\in V} \tilde{f}(s,t,r)\, footprint_h(u-s, v-t) \tag{2.5}$$

We define a projection function $\tilde{p}_w$ by projecting $\tilde{f}$ on the image plane *P* along the view direction *w* as in equation 2.6:

$$\tilde{p}_w(u,v) = \sum_r \tilde{f}(u,v,w)\delta(u-s, v-t, w-r) \tag{2.6}$$

Substituting equation 2.6 into 2.5, equation 2.5 can be reduced as follows:

$$\begin{aligned} I(u,v) &= \sum_{(s,t,r)\in V} \tilde{f}(s,t,r)\, footprint_h(u-s, v-t) \\ &= \sum_{(u,v)\in P} \sum_r \tilde{f}(u,v,w)\delta(u-s, v-t, w-r)\, footprint_h(u-s, v-t) \\ &= \sum_{(u,v)\in P} \tilde{p}_w(u,v)\, footprint_h(u-s, v-t) \\ &= \tilde{p}_w(u,v) * footprint_h(u,v) \end{aligned} \tag{2.7}$$

Here, * is the convolution operator. Since the convolution operation is supported in the OpenGL convolution extension, we can perform all operations in hardware. We project all sample points from the input volume onto the image plane and then perform a convolution operation on the image to create an X-ray image.

2.3 Time Complexity Analysis

The computation time of X-ray convolution can be mainly classified into two parts: 1) the time to project and rasterize the points; 2) the time to convolve the image generated from part 1. Let the time to render and rasterize a point be t_p and the time to do per-pixel convolution be t_c. For an input volume with size of N^3 voxels and an output image resolution of M^2 pixels, the computation time T can be formulated as follows:

$$T = N^3 t_p + M^2 t_c \tag{2.8}$$

In equation 2.8, the point rendering time, t_p, is determined by the graphics hardware rendering capabilities and is a constant for a given graphics hardware. The per-pixel convolution, t_c, varies significantly from the convolution kernel sizes. In general, $t_p << t_c$, however, due to the very large number of samples from the volumetric dataset, the total point rendering time ($N^3 t_p$) is not trivial and can even be much greater than the total convolution time ($M^2 t_c$) for large datasets. This fact has been demonstrated in our experimental results. Note the total convolution time is independent of the data size.

2.4 Implementation

2.4.1 Point-based Rendering and Convolution in OpenGL

The OpenGL primitive, GL_POINTS, is used to render all sample points in the volume onto a rendering target (e.g., a P-Buffer) to generate the projection image according to equation 2.6. After rendering all sample points in a P-buffer, it holds a projection image. To generate the final X-ray image from equation 2.7, convolution is performed by

copying the P-buffer to a target texture with a pre-defined convolution kernel enabled. The target texture thus contains the final X-ray image and is pasted to the frame buffer for visualization. This pasting of the texture adds an extra cost of $t_q M^2$ to the complexity, where $t_q << t_c$. The pseudo OpenGL code to perform point convolution is listed as follows:

(1) Timer.Set();

(2) Render all sample points with GL_POINTS;

(3) glFinish(); Timer.Record() for point rendering time;

(4) Timer.Set();

(5) glEnable(GL_CONVOLUTION_2D);

(6) Copy P-buffer to the target texture;

(7) glDisable(GL_CONVOLUTION_2D);

(8) glFinish(); Timer.Record() for convolution time;

(9) Paste the target texture into the frame buffer.

Figure 2.1 shows the pipeline of this method. The transformed voxels are first projected to a rendering target, called a P-Buffer or a frame buffer object, with the GL_POINTS primitive. The P-Buffer is copied into a shared texture with a convolution filter enabled, and the convolution operation is applied at each pixel. This produces a final X-Ray image as the output of the convolution between its corresponding pixel in the P-buffer and the kernel filter. The X-Ray image in the shared texture is finally pasted into the on-screen frame buffer for visualization. A floating point rendering target is used to

eliminate the alias due to truncation when projecting many points on a same pixel in the P-buffer. Note for our purposes, we use a point size = 1 and anti-alias of the point.

Figure 2.2 shows the projection image in the P-buffer (Figure 2.2a) and its final convolved X-ray image in the shared texture (Figure 2.2b). Figure 2.2b shows the image copied from the P-buffer with convolution for each pixel. Since the P-buffer is on the graphics hardware memory and the texture is shared between the P-buffer and on-screen frame buffer, there is no latency due to the bandwidth limitation for the memory transfer between the CPU and the GPU. All operations are performed on the graphics hardware.

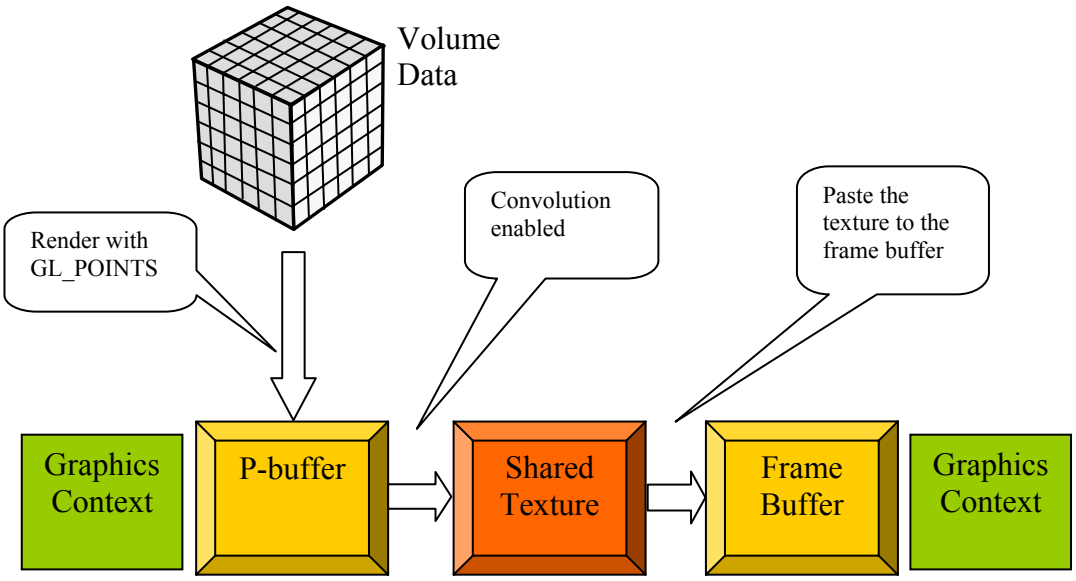


Figure 2.1: The pipeline for point convolution.

(a) (b)

Figure 2.2: (a) The projection image in the P-buffer; (b) The convolved X-ray image.

2.4.2 Memory Management for Large Datasets

To gain high performance for rendering points, we use the vertex buffer object to store the point geometry and intensity information on the graphics hardware memory. The memory for vertex positions must be pre-allocated to be sent down to the graphics hardware when the CPU issues the OpenGL draw primitive array command. One challenge to render large datasets, such as the NIH VisFemale of 512x512x1728, is to allocate the large continuous memory block to hold vertex positions. For example, using a triple of 2-byte short integers to locate vertex positions at regular grid point (*i, j, k*) or the point index of 4-byte integer to locate point position as in [vRHJ*04] requires 2.6GB and 1.7GB, respectively, for the VisFemale. This is beyond the memory capacity for most 32-bit PCs if we also include the memory for the dataset itself.

To render such a dataset on a standard PC, we apply an *instancing* mechanism [NVIDIAc] to render all point primitives. In instancing, a canonical instance of a repeatable set of geometric primitives are grouped into a vertex buffer object and sent down to the graphics memory first. The entire dataset, such as a forest, is represented by repeatedly cloning the instance with different positions, orientations and other attributes like texture coordinates. When rendering a dataset, only the instance-wise attributes and transformation matrices are issued to the GPU, and the vertices in the new instances are re-used from the first instance.

For regularly grid data as in the VisFemale, we can use an m×l×n block as the instance to chop the volume. The entire volume is chopped into blocks of 16×16×16. We only allocate the memory block for a single instance and an intensity array for all points. In our case, we use a simplified *instancing* strategy since all instances have the same orientation and size. A transformation matrix (translation indeed here) is issued for the different instances.

Figure 2.6 shows the rendering result of the VisFemale X-ray image using the *Instancing* mechanism.

Instancing mechanism is also supported in the DirectX 9.0 SDK released in 2004. The DirectX technology developed by Microsoft, is a collection of APIs for developing high-performance, real-time Windows applications. This technique uses two vertex buffers: one to supply geometry data and one to supply per-object instance data.

2.5 Experimental Results and Discussion

All resultant performance data were obtained on a PC equipped with the Pentium4 processor of 3.2GHz, 3.0 GB main memory, and NVIDIA GeForce 6800, 8x AGP based graphics board.

Figure 2.6 shows the rendered X-ray image of size 2048 by 512.

Figure 2.3 plots the performance data for the point projection time vs. the number of sample voxels. The total point rendering time is linearly increased when the dataset size grows. Figure 2.4 shows the point convolution time vs. the image resolutions. The time for convolution solely depends on the image resolution and the convolution kernel size. The bigger the image resolution and the kernel size, the more time for convolution.

The image quality heavily depends on the projection image in the p-buffer from the sample points. Equation 2.7 expects the sample point projection would exactly match their pixel locations in the p-buffer. However, due to the limited resolution of the p-buffer, the ill-projection occurs when the sample point projection is truncated with partial alignment to the pixel in the p-buffer. In our experiment, we can see the alias patterns appear at some view direction when we rotate the volume (see Figure 2.5 bottom).

2.6 Conclusions

We have developed a fast X-ray image generation technique to interactively render very large volumetric datasets. An *instancing* mechanism is used in our method to handle the very large dataset. Our technique shows the timing for X-ray image generation can be decoupled into the point rendering time and the point convolution time. The two parts can

be accelerated using different techniques. Our convolution is performed for all pixels in the image. However, it is not necessary for some datasets with sparse representation in which many empty regions exist. We can develop a partition strategy to perform convolution for the pixels contributing to the final image.

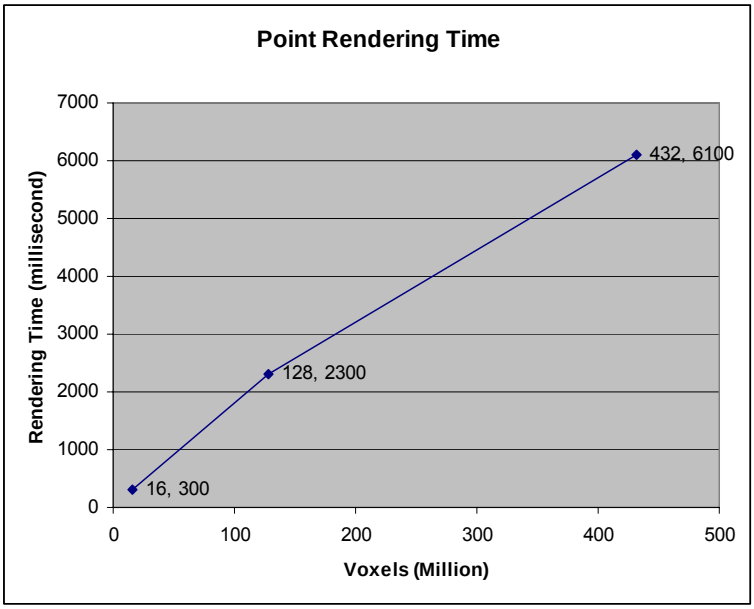


Figure 2.3: Point rendering time vs. the number of voxels.

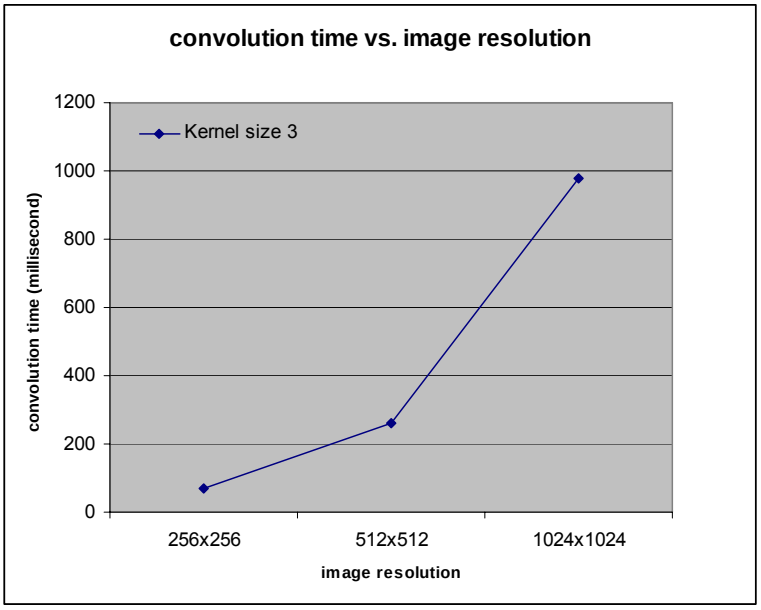


Figure 2.4: Convolution time vs. image resolution.

Figure 2.5: The X-ray image of foot dataset with perfect projection (top) and the aliased image (bottom) due to ill projection.

Figure 2.6: VisFemale X-ray image.

CHAPTER 3

ISOSURFACE-AIDED SLICE BASED VOLUME RENDERING

3.1 Introduction

Using graphics hardware to accelerate volume rendering is continuously exploited by researchers with the advances of new hardware techniques. Cullip and Neumann [CN93] first addressed the capability to render a volume on the 3D texture hardware. Akeley [Ake93] and Cabral et al. [CCF94] described slice-based volume rendering (SBVR). SBVR is a direct mimic of ray-casting, but samples the volume for all rays at once when advancing the rays. The original SBVR slices the whole volume. Engel et al. [EKE01] developed a pre-integrated volume rendering technique for high quality images using multi-texturing. This improves the quality, but not the performance, unless a lower sampling rate can be applied to the volume integration. To improve rendering performance for fairly large dataset, Li et al. [LMK03] split the volume into small bricks. The bricks in empty space are removed and only the non-empty bricks are rendered with SBVR. With the powerful programmability of graphical processing units (GPU) today, many software-based acceleration techniques like empty space skipping and early ray termination [Lev90, DH92, YS93, FS97] can be implemented on the GPU directly.

Krüger and Westermann [KW03] and Roettger et al. [RGW*03] develop their algorithms to perform ray-casting using a pixel shader 2.0 program [Mic02] on the GPU with early ray termination and space-leaping. Krüger and Westermann propose an ingenious encoding of the ray direction and length into floating point render targets. These textures are then used to determine where to sample the 3D texture (volume). The early z-culling feature on the latest graphics hardware makes early ray termination possible in their algorithms.

In a typical slice-based volume rendering, the volume is sliced by the object-aligned or image-aligned planes (see Figure 3.1). These planes are rendered in a back-to-front or front-to-back order, textured by the 3D texture (volumetric dataset) during rasterization, and finally composited into the frame buffer to generate the final image. A main drawback of SBVR is that, for each slice during rasterization, all fragments are sampled from the 3D texture even though some fragments do not contribute to the final image at all. This greatly reduces the rendering speed, especially when a complex fragment shader including lighting or high-order gradient computation is employed. This is very inefficient since the empty space usually occupies more than one-third of many volumetric datasets.

In this chapter we present an isosurface-aided hardware acceleration technique for slice-based volume rendering (iSBVR). The acceleration is based on the early z-culling feature provided by the latest consumer level graphics hardware. Given a transfer function, we can analyze it to determine values where the resulting opacity is completely opaque. Extracting iso-contours corresponding to these values provides a blocking

surface, where any sample of the volume along the ray that is behind (or within) this surface is not visible. Isosurfaces can also be extracted corresponding to any minimal thresholds in the specified transfer function (i.e., where the transfer function goes to zero opacity). These surfaces do not block the rays as in early ray termination, but can provide a simple space-leaping as we will show later in the chapter. More importantly, the minimal isosurfaces can be used to flag areas on the screen where the ray passes entirely through volume without hitting any values that would contribute to the volume integral. We call these rays empty rays and our algorithm provides an efficient solution for *empty ray removal.* It should be noted, that these isosurfaces are rendered only to initialize the z-buffer. Nothing is ever skipped, but with early z-culling enabled, the hardware quickly processes these areas resulting in a substantial performance improvement.

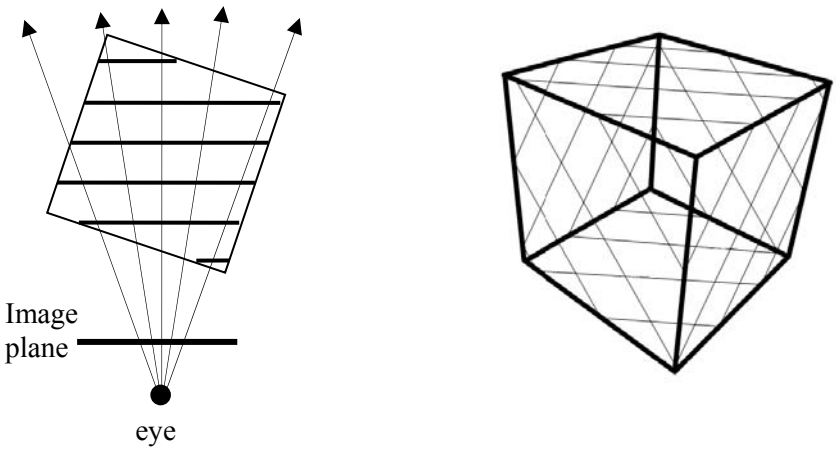


Figure 3.1: The proxy geometries of image-aligned slicing planes. (a) 2D diagram of slice planes. (b) The slicing planes intersecting with the volume box.

3.2 Early Z-culling for Volume Rendering

A key observation of brute-force texture-based volume rendering is probably the sheer number of fragment and pixel operations which do not contribute to the final image. This problem becomes more serious with a complex fragment shader, which includes texture accesses for volume sampling, transfer function lookups, gradient and lighting calculations, and blending operations.

Effective utilization of the early z-culling feature on graphics hardware is the impetus for our isosurface-aided acceleration technique. The key criterion here is that the z-buffer must be set up properly such that only fragments on the slicing plane that contribute to the final image can pass the depth test. By means of early z-culling, the fragments that do not contribute to the final image will exit from the graphics processing pipeline immediately. As pointed out by Krüger and Westermann [KW03] and our own experiments, this early termination greatly reduces the rendering time, particularly when complex shaders are desired.

Assume that for each pixel a z-value can be determined such that further samples will be occluded. Ideally, we would like to set our z-buffer to these values. Furthermore, if the ray passes entirely through empty space or air, then the processed fragments can be skipped. Our goal is to set the z-value to the front of the volume for these rays. By setting the z-buffer as such, the rendering speed can be benefited from the early z-culling feature of modern graphics hardware.

A two pass rendering process is used in most games containing complex shaders. In the first pass, a simple shader is performed to set up the z-buffer. If the color frame buffer is not being changed, newer hardware can actually render this pass twice as fast. In the second pass, a final complex shader is performed. Theoretically, this shader is performed for all fragments. Any fragments which fail the depth test, are then simply discarded. The early z-culling feature of the hardware performs the depth test first, and only if it passes does the resulting complex shader get processed. Hence, only the visible fragments are rendered in the second pass (note, the hardware is a little more complicated than this). The remaining *non-effective* fragments are occluded and the shader on them is skipped.

For direct volume rendering, things are much more complex, as opaque surfaces (or positions) are not clearly defined. Kruger and Westermann [KW03], developed a ray-caster in the graphics hardware. An early ray termination was implemented using the early z-culling feature, by processing the rays in slabs. After each slab, a rendered texture from the opacity buffer would be examined in a fragment shader, and pixels which were fully opaque would have their z-values set to the current slab position. Roettger, *et al* [RGW*03], do a similar thing, with a slab width of 4 samples and an occlusion query test for region of entire image termination. Newer hardware, such as nVidia's 8800 allows for better looping and branching [NVIDIAb] and a true implementation of ray-casting with early ray termination. This does not need, nor use the early z-culling feature of the hardware.

3.3 Time Complexity Analysis

We classify all fragments, F, into either affecting the volume integral or not. Those affecting the integral will need to execute their corresponding fragment shader. Thus, we have F_c fragments for which a complete and potentially complex shader needs to execute, taking on average T_c time per fragment. Our goal is not to remove or ignore any superfluous fragments, but to reduce their shader time to the minimal execution time T_z by the early z-culling. Essentially, there are three main computational parts for a two pass volume rendering:

1. Time to set up the z-buffer in the first pass: $F_s * T_s$;
2. Rendering fragments that are discarded by early z-culling in the second pass: $(F-F_c) * T_z$;
3. Fragments rendered with the complex shader in the second pass: $F_c * T_c$.

The above three parts lead to Equation 3.1 as a computational model for the 2-pass volume rendering time, $T_{2\text{-}pass}$, with a maximal potential speedup, δ, given in equation 3.2. This is provided we can control the hardware to only execute the complex shader on the affective fragments.

$$T_{2-pass} = F_s * T_s + (F - F_c) * T_z + F_c * T_c \tag{3.1}$$

$$\delta = \frac{F * T_c}{F_s * T_s + (F - F_c) * T_z + F_c * T_c} \tag{3.2}$$

Where,

F: the total number of fragments generated from the volume;

F_s: the number of fragments to set up the z-buffer in the first pass;

F_c: the number of fragments fed into the complex shader in the second pass;

T_s: the operation time of a simple shader to set up the z-buffer in the first pass;

T_z: the operation time for a fragment discarded by the early z-culling (with no fragment program at all) in the second pass;

T_c: the operation time of a complex shader to render the final image in the second pass.

For slice-based volume rendering, each slice is rendered twice. In the first pass, a simple shader is applied to modify the z-buffer if a pixel reaches opaque in the opacity buffer. This slice is rendered again by a complex shader with early z-culling enabled. In this case, the number of fragments, F_s, in the first pass to set up the z-buffer equals to the total number of fragments, F. In general, the simple shader time, T_s, is close to T_z and we will use T_s to approximate T_z in our later discussion. Equation 3.1 and 3.2 can thus be approximated by:

$$T_{2-pass} = F * T_s + (F - F_c) * T_s + F_c * T_c \tag{3.3}$$

$$\delta = \frac{F * T_c}{F * T_s + (F - F_c) * T_s + F_c * T_c} \tag{3.4}$$

The simple shader time T_s is fixed for a given graphics hardware, and the complex shader time T_c varies upon different shaders. Let the fragment culling rate be $\alpha = (F - F_c) / F$ and the simple shader speed-up be $\gamma = T_s / T_c$. Equation 3.4 can then be simplified as:

$$\delta = \frac{1}{\alpha\gamma + 1 - \alpha + \gamma} \tag{3.5}$$

Two pass rendering is beneficial when the speedup, δ, is greater than 1. Substituting $\delta > 1$ into Equation 3.5, we obtain our desired property:

$$(1-\gamma)(\alpha+1) > 1 \tag{3.6}$$

Inequality 3.6 describes for a given shader (γ is fixed), how many fragments must be occluded to gain a speedup in any two pass SBVR of the volume. For example, if a more complicated SBVR shader has γ = 0.2, the fragment culling rate, α, must be greater than 25% to gain a speedup. The goal of the next section is to provide a fast and efficient scheme for setting the z-buffer such that F_c is as close to the number of affecting fragments, F_a, as possible.

3.4 Isosurface-aided Hardware Acceleration

While our algorithm will work with any opacity-based volume shader, it relies on the mapping from function values to opacities (i.e., the transfer function), to have certain characteristics. Not like the regular early ray-termination which occurs when the accumulated opacity from the transfer function reaches to the maximum of one, *pseudo* early ray-termination will only occur when the transfer function reaches a maximum opacity of one or other value selected as opaque opacity. Space-leaping and empty ray removal, provide greater benefits when the transfer function contains regions with zero opacity. In other words, if opacity does not equal zero, there will be no empty space that we can remove. If the transfer function does not have either of these properties, then it should be noted that there is no overhead associated in the volume rendering due to this technique.

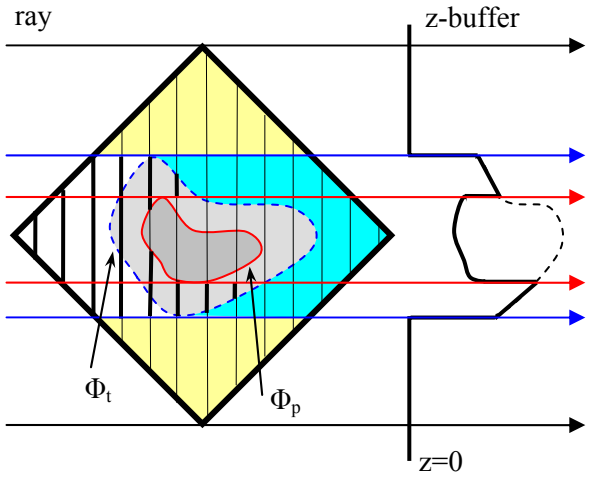


Figure 3.2: The back faces of isosurfaces Φ_t and the front faces of isosurface Φ_p are rendered with parallel projection and their corresponding z-buffer (right). Only the slices in bold pass the depth test and contribute to the final image.

For simplicity in the discussion, we will assume we have only two isosurfaces, Φ_t and Φ_p, given by a boundary threshold where the opacity goes from zero to a non-zero value and an opaque threshold where the opacity reaches one. A very simple example is given in Figure 3.7e. In general, several iso-values can be used, albeit at a potential rendering cost. The resulting isosurfaces are extracted in either a pre-processing stage, or whenever the transfer function is changed. Figure 3.2 illustrates a cross-section of the volume rendering process containing an opaque iso-contour (red solid line), and the boundary iso-contour (blue dashed line). For discussion, we will also assume that all isosurfaces are closed for now, and that the interior of these closed isosurfaces have values greater than the isovalue. This former assumption will be discussed and removed in later sections. If the later assumption is violated, then only an opaque surface will be

visible. Note, these assumptions are on the opacity values, not the actual function values.

3.4.1 *Pseudo* Early Ray-Termination

Clearly, any fragments which lie behind another fragment which is opaque, will not contribute to the volume integral. If we set the z-buffer to the front-faces of this isosurface, we will enable early z-culling on the remaining fragments. This is not true early ray-termination, in that the ray could reach maximal opacity long before reaching an opaque isosurface. This region is depicted by the depth buffer between the two rays (red color) in Figure 3.2. The main steps to initialize the z-buffer for pseudo early ray-termination are thus:

- **z-Buffer Initialization:** for early ray termination.
    1. Disable the output to the color buffer;
    2. Set the depth function to GL_LESS (the default);
    3. Render the **front** faces of Φ_p.

This simple process provides a speed-up from 30% to 50% in our tests.

3.4.2 Empty Space Leaping

A typical volume will have many pockets of empty space, some between the eye and the volume material, some within the volume and some between the volume and the background. Culling away all of these fragments is a challenging research question. Space-leaping typically concentrates on removing the material between the eye and volume. This corresponds to the first crossing of the ray with the minimal iso-contour value or Φ_t surface. We can set the z-buffer to these crossing, by rendering the front faces

of the isosurface. Early z-culling can then be achieved by using a GL_GREATER depth test on the fragments. The main steps to initialize the z-buffer for space-leaping are thus:

- **z-Buffer Initialization**: for space-leaping.
    1. Disable the output to the color buffer;
    2. Render the front faces of Φ_t.
    3. Set the depth function to GL_GREATER.

3.4.3 Combined Space-Leaping and Early Ray-Termination

Early ray-termination requires a GL_LESS test, while space-leaping, a GL_GREATER test, seeming to preclude the use of both accelerations in the same rendering. Space-leaping is usually associated with setting the initial sample location for a ray. We can reverse the ray direction, and test if the current sample location is the last contributing sample along the ray. Here, we remove the material between the volume and the background, and call this exit-based space-leaping. This corresponds to the last crossing of the ray with the minimal iso-contour value or Φ_t surface. This is the region between the two rays (blue color) in Figure 3.2. The main steps to initialize the z-buffer for exit-based space-leaping are thus:

- **z-Buffer Initialization:** for exit-based space-leaping.
    1. Disable the output to the color buffer;
    2. Render the front faces of the volume's bounding box.
    3. Set the depth function to GL_GREATER;
    4. Render the **back** faces of Φ_t.
    5. Set the depth function back to GL_LESS.

Now, to combine this with the early ray-termination, we simply need to perform the initialization for exit-based space-leaping before the initialization for early ray-termination. After the exit-based space-leaping initialization, the z-buffer will either have values corresponding to the front faces of the bounding box, or the last surface of the minimal iso-value. For our assumptions with closed iso-contours, the early ray-termination surfaces, Φ_p, will project only to areas already covered by the isosurface, Φ_t. Since the z-buffer was pushed away from the cube faces in these regions, the early ray-termination initialization will pull these back towards the viewer.

3.4.4 Empty Ray Removal

For sparse values, many rays do not intersect any meaningful data in the volume. The rays end up being set to the background color. This implies that a ray never crosses through the minimal isosurface, Φ_t, (and by the closed assumption the opaque isosurface, Φ_p, as well). Early z-culling for the fragments in these areas will work if the z-buffer is set to a minimal value (zero or the front faces of the volume). The exit-based space-leaping algorithm above actually accomplishes this already. The region (in yellow color) in Figure 3.2 represents the empty rays, and the resulting z-buffer is set to zero in this case. Combining the empty ray removal and the exit-based space-leaping provides a substantial speed-up between 200%-300%.

3.4.5 Culling Efficiency

Our final, and significant, result is that only the fragments on the bold portion of the slices in Figure 3.2 will pass the depth test and execute any complex shader associated

with them. The performance improvements from both empty space skipping and early ray termination, achievable with most software-based ray-casting algorithms [Lev90, DH92, YS93, FS97], are now accomplished in the context of slice-based volume rendering by leveraging the early z-culling feature of modern graphics hardware. We still have some fragments which do not contribute to the final image, but pass through the z-cull operation. Hopefully this set is greatly reduced. The actual results will be data set and transfer function dependent. To render the volume, we simply need to turn on depth testing as usual and process the volume slices. The algorithm works equally well using a back-to-front or a front-to-back slicing order. The volume is rendered as usual:

- **Volume Pass:** render the proxy geometries.
    1. Enable the output to color buffer;
    2. Set the depth function to GL_LESS;
    3. Enable the fragment or volume shader;
    4. Render the proxy geometry.

Figure 3.7 shows the isosurfaces and the resultant z-buffer after the initialization passes. A black or darker value indicates $z = 0$ while a white value indicates $z = 1$. Darker values indicate the depth is closer to the eye.

In order to characterize our algorithm, we need to consider the rendering time, T_a, from the two initialization passes (Equation 3.7) and the rendering time, T_m, for the volume shader (Equation 3.8). Any resulting speedup is characterized by Equation 3.9.

$$T_a = F_t * T_s + F_p * T_s \tag{3.7}$$

$$T_m = (F - F_c') * T_s + F_c' * T_c \tag{3.8}$$

$$\begin{aligned} \delta &= \frac{F * T_c}{T_a + T_m} \\ &= \frac{F * T_c}{F_t * T_s + F_p * T_s + (F - F_c') * T_s + F_c' * T_c} \end{aligned} \tag{3.9}$$

Where,

F_t: the total fragments generated from the back faces of Φ_t ;

F_p: the total fragments generated from the front faces of Φ_p;

F: the total fragments generated from the volume;

F'_c: the number of fragments fed into the complex shader in iSBVR.

Obviously, if the time it takes to render the isosurfaces approaches the volume rendering time, and potential speed-ups in the volume rendering are lost. The next section examines the issues associated in generating and rendering the isosurfaces.

3.5 Isosurface Extraction

For a typical dataset of 256^3, there could be more than a million triangles on the isosurface (see Figure 3.6a) for a reasonable transfer function. The rendering time for this large number of triangles offsets any speedup from early z-culling. To reduce the isosurface rendering time, an octree is generated from the underlying volume to extract the isosurface. Each octree node contains a min-max value pair representing the minimal and maximal voxel values it includes.

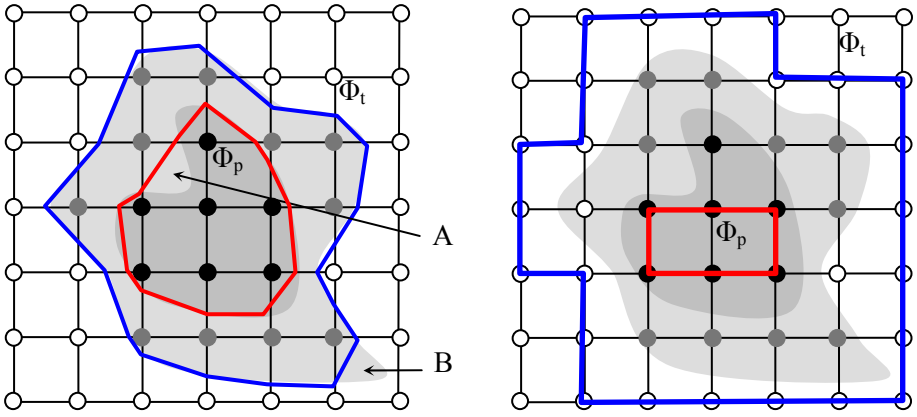


Figure 3.3: Isosurfaces and their reduced form in cube faces. Left: isocontouring, Φ_t and Φ_p. Right: Φ_t is inflated to the outer faces of the cubes containing it. Φ_p is shrunk to the outer faces of the inter cubes.

However, when generating the isosurface from the octree using the maximal value in the octree node, the iso-suface, Φ_t, may occlude some voxels even though their values are greater than the iso-value since the voxel with the maximal value is not necessarily the vertex in the node for iso-contouring. The voxel labelled **B** in Figure 3.3 (left) shows this case. Similarly, the isosurface, Φ_p, may contain the non-opaque area as **A** in Figure 3.3 (left). The holes in Φ_p will produce serious aliases since the rays will stop at the Φ_p due to early z-culling. To solve these problems, the isosurface, Φ_t, is inflated to fill the outmost cubes that containing it and the Φ_p is shrunk to the maximal set of the cubes completely included inside Φ_p. Figure 3.3 (right) shows the reduced isosurfaces for Φ_t and Φ_p. Figure 3.4 show the bonsai dataset rendered by the original isosurface and its reduced isosurface. The holes in the original one (Figure 3.4a) have been removed in Figure 3.4b.

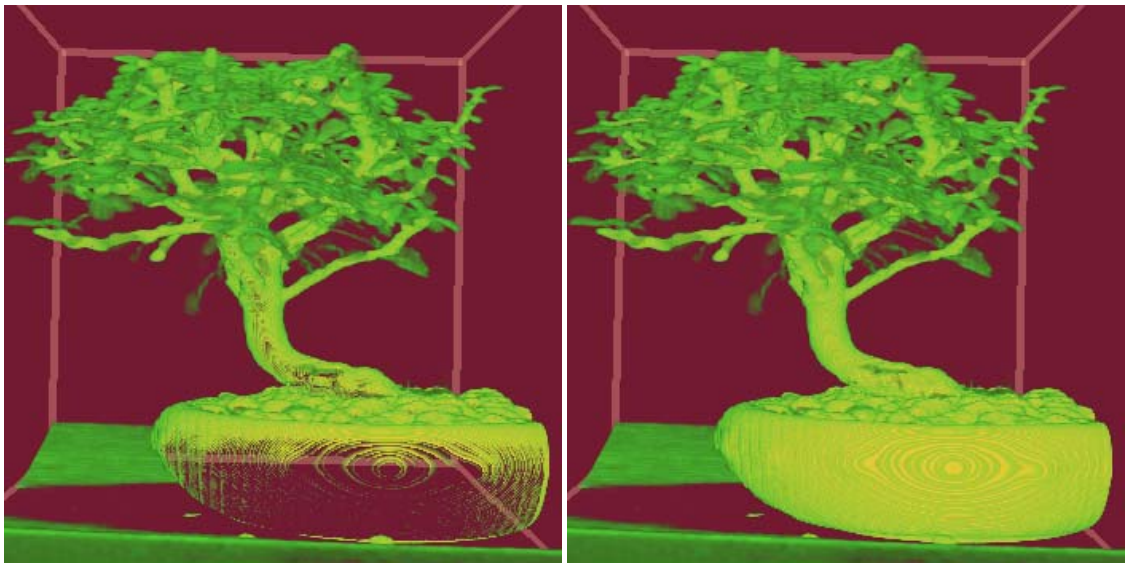


Figure 3.4: (a) Generated from the original isosurface. There are holes in the image due to the incorrect occlustion. (b) Generated from the reduced isosurface with holes removed.

3.5.1 Isosurface for Empty Space Leaping

As shown in Figure 3.2, only the back faces of Φ_t are used to set up the z-buffer for front empty space leaping. Thus, we can generate a set of cubes which contain the manifold of the isosurface, and render these cubes with back faces and with GL_GREATER for depth testing.

The isosurface must be closed to correctly set up the z-buffer in the two initialization rendering passes in section 3.4.1 and 3.4.3. Otherwise, there are the undesired z-values from the front faces for the open area in the z-buffer that will incorrectly occlude the fragments in the final image. However, if the value of the voxel on the volume boundary is greater than the input isovalue, the final output surface will be

open around such voxels. To create the close isosurface, the cube on the volume boundary is also added to the cube set if it is inside of the isosurface. The algorithm to create the cube set is listed as following:

Input: the cubes of the octree
Output: the cube set *S* containing Φ_t.
1) Set the cube set S = $\varnothing$;
2) For each cube *d* in the octree of the volume
3) If *d* contains isosurface
4) S = S $\cup$ {d};
5) Else if d is inside the isosurface and on the volume boundary
6) S = S $\cup$ {d};
7) Endif
8) Endfor
9) Return S.

3.5.2 Isosurface for Early Ray-Termination

To create isosurface, Φ_p, for the opaque values, we use the minimal value in each octree node. When the octree node size is big (accordingly low resolution with respect to the original volume), some parts of the isosurface could be missed. This is not desired since we want more fragments can be rendered to set up the z-buffer for early ray termination. This problem becomes more serious, especially for medical datasets in which the skull or thin bones cannot contain a complete cube from the octree node. Figure 3.5 shows a 2D diagram where the iso-contour shrinks when iso-contouring using the minimal value from the min-max pair in the octree node. Figure 3.6 shows the isosurface of Φ_p from a Siemens CT head dataset. It shrinks drastically when octree node size increases from 1 to 8. There are almost no pixels to be set with the z-values for early ray termination in the

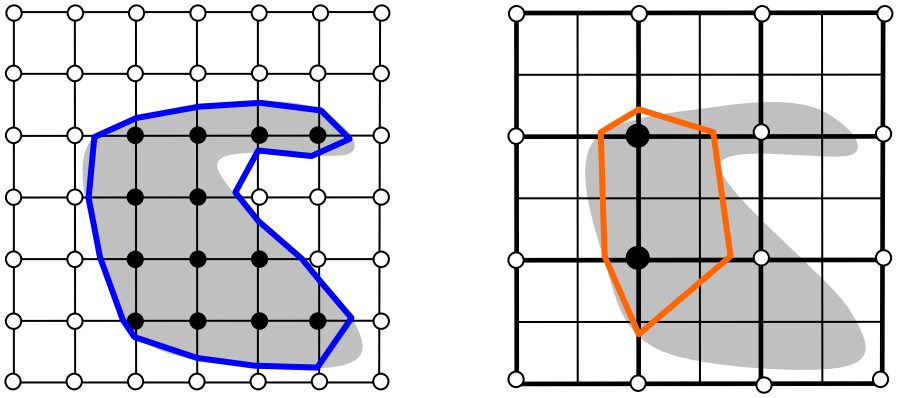
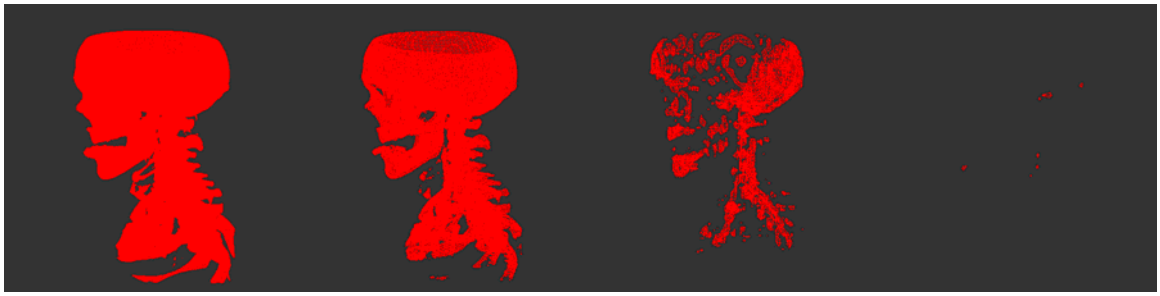


Figure 3.5: Left: iso-contouring for 7x7 grid. Right: the grid is generated from left with quad-tree node of 2x2. The vertex value is determined by the minimal value of each 2x2 node from the left grid.



(a)	(b)	(c)	(d)
original	*2x2x2 node*	*4x4x4 node*	*8x8x8 node*
1,059,856 Triangles	*213,416 Triangles*	*16,792 Triangles*	*80 Triangles*

Figure 3.6: The isosurface shrinks drastically when using the minimal value to perform contour on different octree levels.

initialization pass if using octree node of 8x8x8. In our experiments, the octree node of 2x2x2 for Φ_p provides the good balance between the overhead to rendering the triangles on Φ_p and the benefit from the early ray termination.

To shrink the isosurface as shown in the Figure 3.3 (right), we shrink the cube set containing isosurface, Φ_p, until it only includes the cubes which are completely inside Φ_p. The algorithm for creating such cube set, *T*, is as following:

Input: the cube set E containing Φ_p
Output: the shrinking cube set T
1) Set the cube set T = $\varnothing$;
2) Repeat each cube *d* in E
3) E = E - {d};
4) For all d's neighbour n_j along shrinking direction
5) If n_j is completely inside Φ_p and $n_j \notin T$
6) T = T $\cup$ { n_j };
7) Else
8) E = E $\cup$ { n_j };
9) Endif
10) Endfor
11) Until E = $\varnothing$
12) Return T.

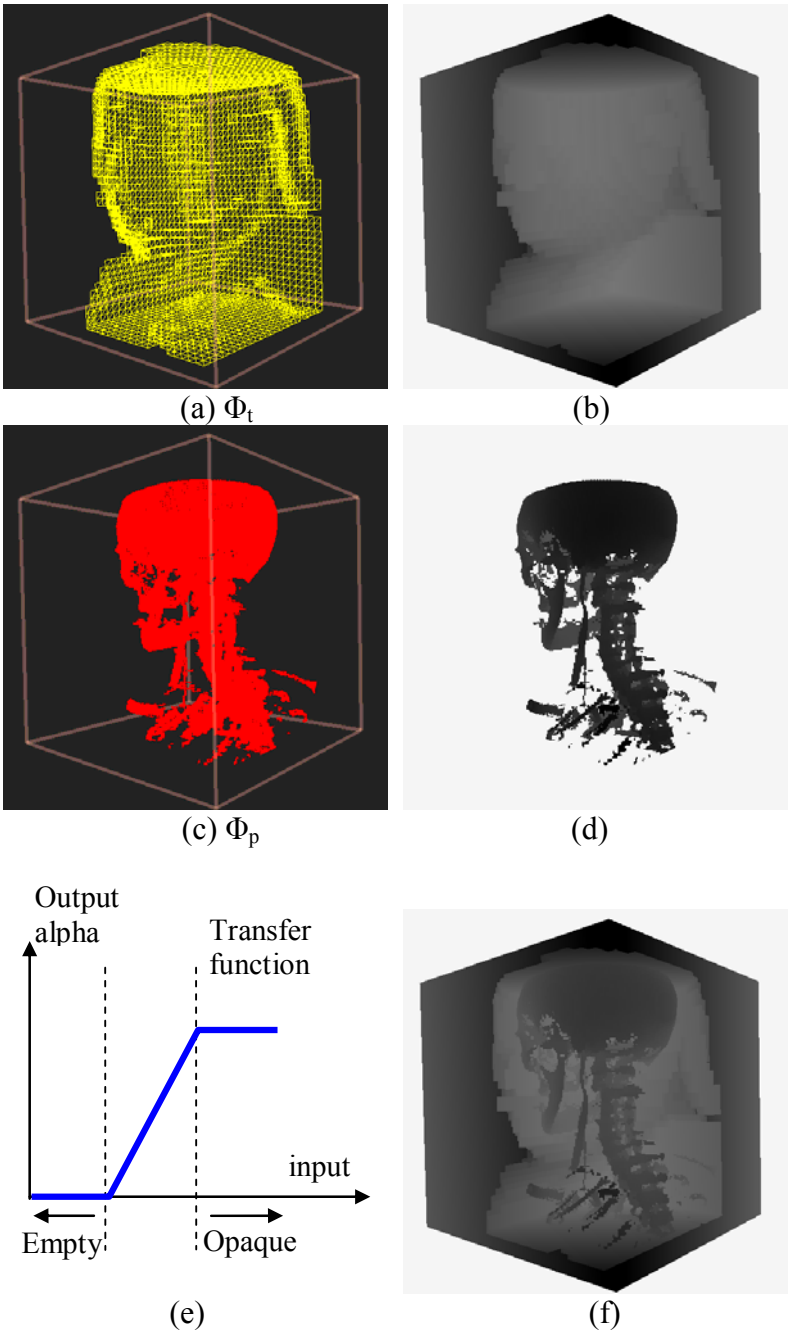


Figure 3.7: (a) The back faces of isosurface Φ_t; (b) The z-buffer after rendering the isofurace in (a); (c) The front faces of isosurface Φ_p; (d) The z-buffer after rendering the isosurface in (c); (e) the transfer function for two isosurfaces; (f) The z-buffer is rendering after the two initialization passes. Note: the values in the z-buffer images (right column) are rescaled to highlight the difference.

3.6 Results and Discussions

All performance data were obtained on a standard PC equipped with an ATI 9800 pro graphics card with 128 MB video memory. All four datasets are of 256^3 in our tests and the slice spacing was set to the voxel interval distance. The resultant imagery from a gradient-based shader is shown in Figure 3.8. The performance results are drawn in Figure 3.9. Our results show that we obtain on average 2 to 3 times speedup against the brute-force SBVR. Figure 3.9 shows that the performance of early ray termination (ERT) for the CT head dataset II is greatly reduced due to the large fuzzy area in the volume (Figure 3.8b), while the performance of empty ray removal (ERR) for aneurism dataset is significant due to the sparse representation of the dataset (see Figure 3.8c).

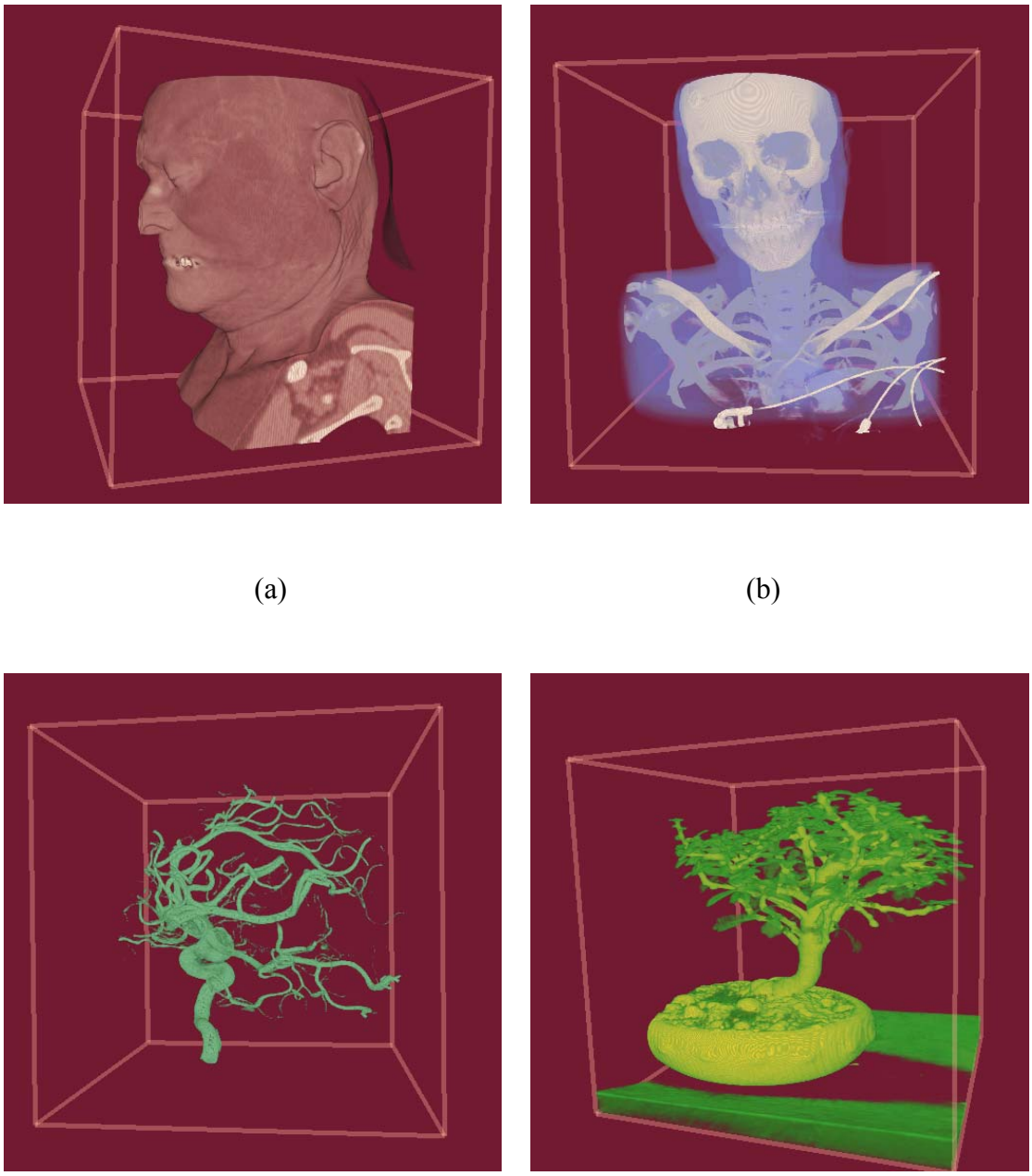
The slices in front of any front face of Φ_t are still fully rasterized and executed by the fragment program (see Figure 3.2) since their depths are always less than the pre-rendered depth value in the z-buffer. This problem can be solved by rendering each slice with one more pass as in [KW03], in which a simple shader is performed to modify the z-buffer to occlude pixels reaching opaque. On the other hand, considering the large number of slices, the overhead of the additional rendering passes for all the slices partly offsets the performance improvement. If OpenGL would support a depth band-test with dual z-buffers, this would further improve the current frame-rates by culling all fragments outside of the two z-buffers without additional overhead (assuming the additional depth test is free). The newly introduced GL_DEPTH_BOUNDS_TEST_EXT provides a similar but much simplified function, in which a user specified depth range test between [0..1] is applied to fragments in addition to the normal depth test. This OpenGL

extension can help the performance improvement of the iSBVR by excluding all the voxels beyond the depth bound.

The iso-contouring is performed in either a pre-processing stage, or whenever the transfer function is changed. Since we only apply iso-contouring to volume octrees (8x8x8 and 2x2x2 nodes for Φ_t and Φ_p, respectively), this greatly reduces the number of cells for iso-contouring. We can still obtain interactive rendering speed when changing transfer functions. In our experiments, the timings for isosurface extraction are 10 ms and 210 ms for Φ_t and Φ_p, respectively. The other well-studied accelerated isosurface extraction techniques [SHLJ96, vRHJ*04] can be used to further enhance the performance.

3.7 Conclusions

By means of the early z-culling feature, we have developed an isosurface-aided hardware acceleration technique for slice-based volume rendering to gain the improved frame-rates of two to three times. The advantages of early z-culling become more pronounced for hardware accelerated volume rendering. This isosurface-aided acceleration can be easily fit into the other existing GPU volume rendering pipeline like Krüger and Westermann's GPU-based ray caster [KW03] and the pre-integrated volume rendering [EKE01].



(a) (b)

(c) (d)

Figure 3.8: All images are of resolution by 512x512. (a): CT head dataset I (256^3); (b): CT head dataset II (256^3); (c): aneurism dataset (256^3); (d): bonsai dataset (256^3).

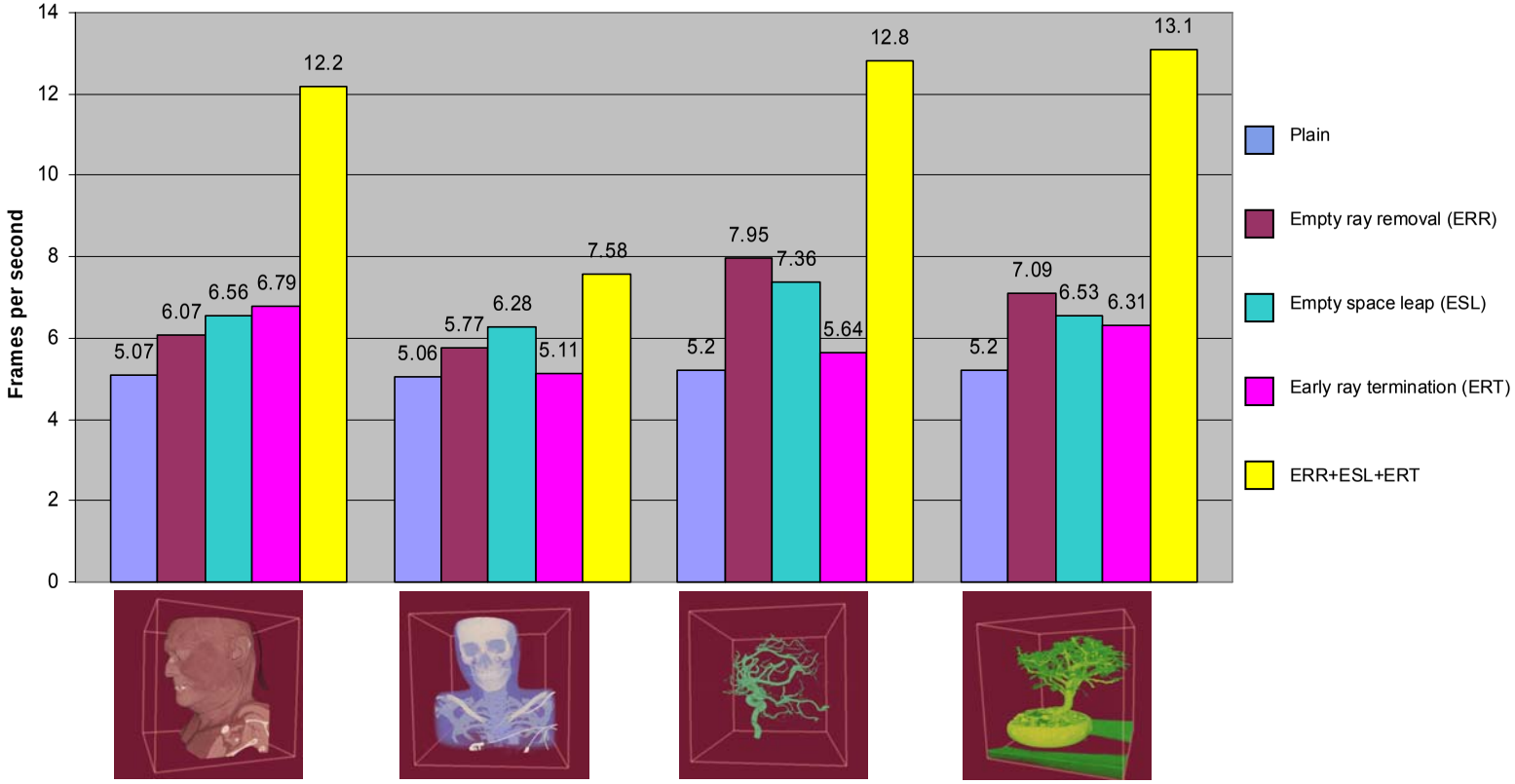


Figure 3.9: The rendering FPS with different acceleration techniques for four datasets using iSBVR.

PART II

GRAPHICS HARDWARE VOLUME SHADERS

CHAPTER 4

TEXTURE SPLAT SHADER

4.1 Introduction

Texture mapping hardware has been used in splatting algorithms [LH91, CM93] to visualize scalar and vector field volumetric datasets. Interactive rates are only achievable with rather small datasets. With the development of advanced graphics hardware in recent years, it is worthwhile to develop a more efficient texture splat shader for fast interactive and dynamical flow volume rendering [XC04].

Crawfis and Max [CM93] explore anisotropic textured splats to represent multi-variate vector fields by using a large set of textures adding different length vector icons to the reconstruction kernel texture. King et al. [KCR99] and Wei et al. [WLMK02] develop another set of textured splats with gaseous details to represent amorphous volumes like fire. They both generate a table of textures with different phase-shifts to achieve animation for vector fields and gaseous volumes.

In OpenGL, hardware accelerated splatting for vector fields is achieved with the following steps:

1. Create a texture image from the reconstruction kernel function as the splat footprint;
2. Create a set of anisotropic textures for the vector field, fire or smoke rendering;
3. Create a quadrilateral which is centered about the voxel location for each voxel in the volume;
4. Sort all voxels along the view direction;
5. Reorient each voxel quadrilateral to be perpendicular to the viewing ray;
6. Select the appropriate texture based upon the vector field direction at the current voxel;
7. Render the quads with texture mapping in a back-to-front order.

In this chapter, we present a dynamic, multi-glyphic textured splatting technique to render multi-variate flow volumes with the support of vertex shaders on graphics hardware. We implement a vertex program, using the OpenGL multi-texture and register combiner extensions to construct anisotropic textured splats, which can represent vector fields, and to dynamically visualize the flow volume. We achieve the animation effect in a very efficient way, while using only one or two small textures. In addition, we provide multi-glyphic textured splats to classify and visualize the vector field.

4.2 Textured Splats

For vector field representations, we embed directional icons or glyphs into the texture used for the footprint integration [LH91, Wes89, Wes90]. Two textures, one for the scalar kernel map and one for anisotropic vector icons, are generated with only an alpha channel. To represent the vector field correctly, the quadrilateral splat must be rotated

according to its vector field direction in eye space. The view-dependent reorientation of the splat is performed in the vertex program of vertex shader [Wyn], which will be discussed in the next section. This adds an additional complexity to the vertex program which normally only handles the transformation and lighting stage of the OpenGL pipeline [LKM01]. The splat is then rotated according to its projected vector direction in eye space, such that the mapped anisotropic vector icon aligns with its projected vector direction.

Specifically, we investigate vector field animation, multi-variate color-coding of the vector icons and multi-glyphic textured splats, and vector icon foreshortening due to its projection on the plane orthogonal to the view direction in this section.

4.2.1 Multi-Variate and Multi-Glyphic Textured Splats

Representing data sets with both scalar values and vector icons offers us more cues and insights about the relationship between these fields [CM93]. Figure 4.1 shows an image reprinted from [CM93], where the percent cloudiness is rendered using a traditional scalar field volume renderer via splatting, and the wind velocity is rendered using the textured splats icons. Here, the vector icons are color-coded by an independent variable. In this case, the altitude was used to provide additional positional information. To accomplish this, we use a BLEND operator to create a splat's color as a dissolve between the quad color $(R_{pri}, G_{pri}, B_{pri})$ and a vector icon *secondary* color $(R_{\text{sec}}, G_{\text{sec}}, B_{\text{sec}})$. The splat's resulting color is represented as:

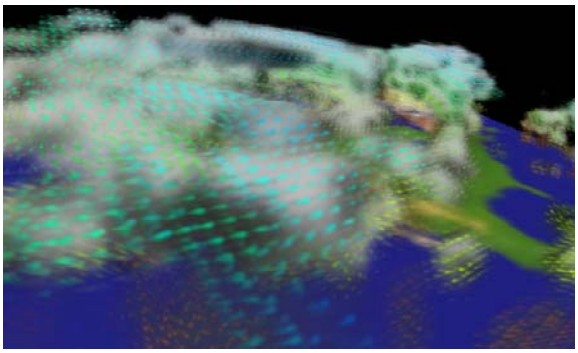


Figure 4.1: Percent cloudiness and wind velocities. The wind velocities are color coded by altitude. Courtesy to Roger Crawfis.

$$
\begin{aligned}
R &= R_{pri} * (1 - A_{tex1}) + R_{\sec} * A_{tex1} \\
G &= G_{pri} * (1 - A_{tex1}) + G_{\sec} * A_{tex1} \\
B &= B_{pri} * (1 - A_{tex1}) + B_{\sec} * A_{tex1} \\
A &= A_{pri} * ((1 - A_{tex1}) * A_{tex0} + A_{tex1}))
\end{aligned}
\tag{4.1}
$$

Here, A_{tex0}, A_{tex1} are the alpha channels of the reconstruction kernel texture and the vector icon texture, respectively. The splat's color is then attenuated by its opacity and composited into the final image with the blending function *glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)* in a back-to-front rendering order [MHB*00, Max95].

Figure 4.2 shows the blending operation between the two textures used by the hardware [Spi]. Both the reconstruction kernel map and the vector icon are generated as only one channel (alpha channel) textures and they are assigned as texture unit 0 and texture unit 1 respectively. The primary color is assigned using the voxel intensity with a proper transfer function, while the secondary color is assigned as the vector icon color. Figure 4.2 demonstrates the register combiner diagram to compute the splat's color and

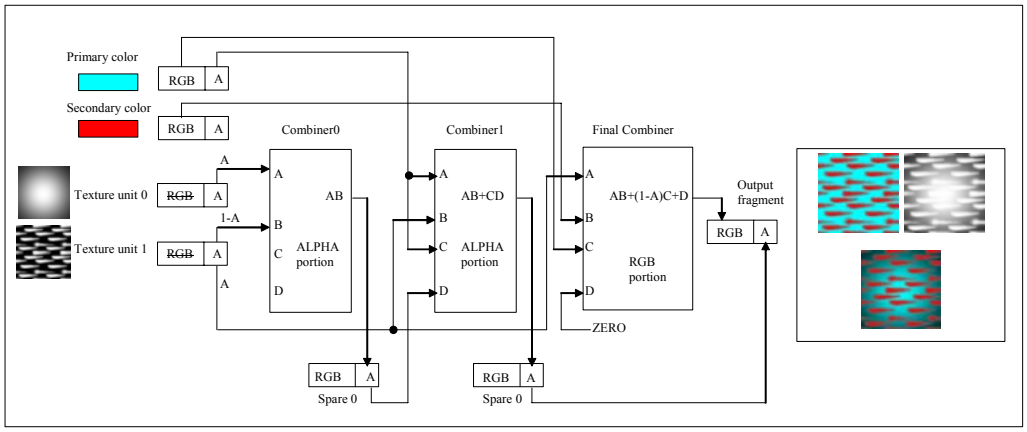
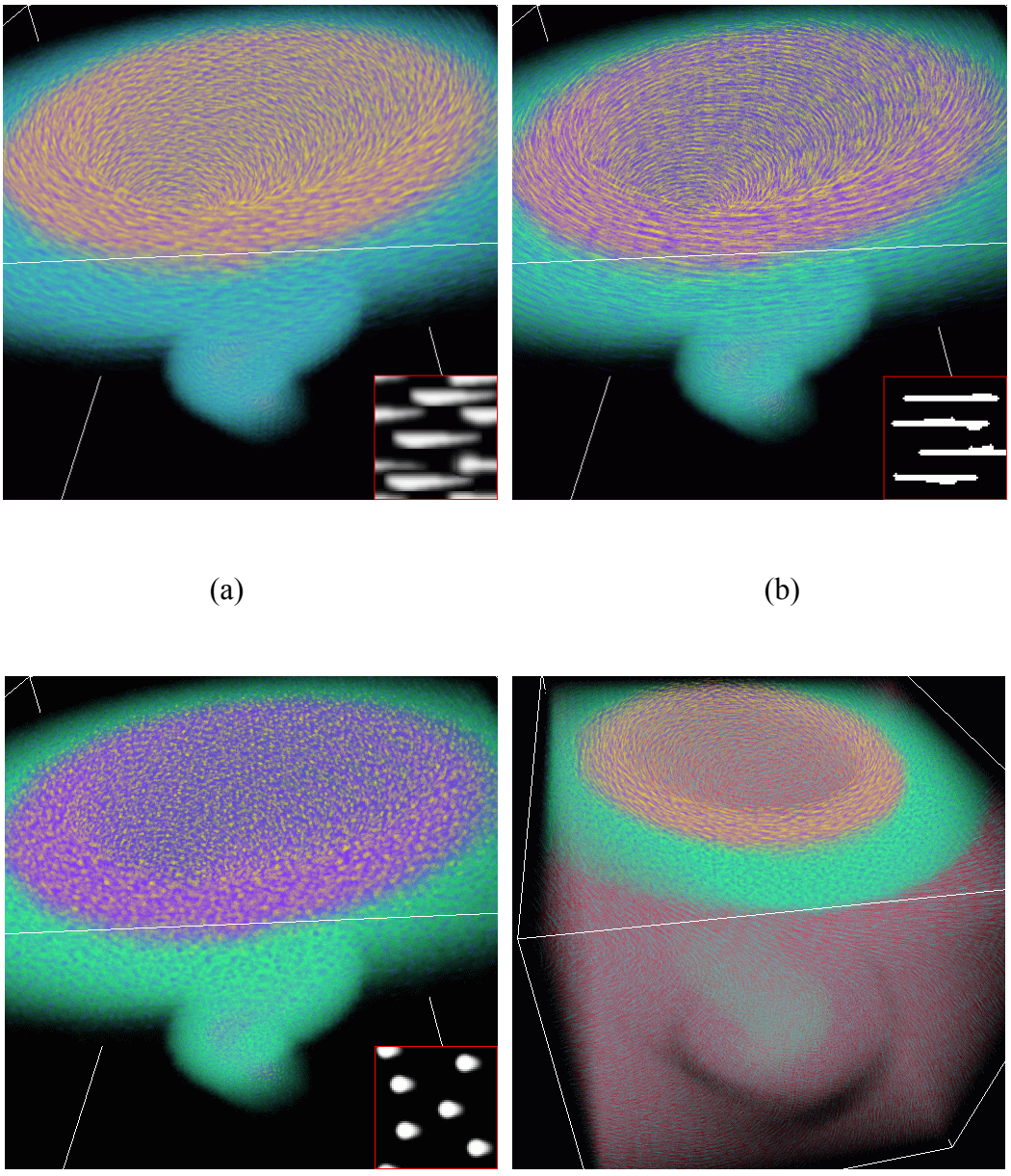


Figure 4.2: the register combiner diagram for producing the splat color using our BLEND equation.

opacity. Its final color (modulated by its opacity) is also presented at the bottom right.

In the above discussion, we use color to code the vector tensor. To obtain better insight of vector fields, we can use different vector icon textures to represent the flow in the same volume rendering (multi-glyphic textured splats). Each textured splat is associated with a specific vector icon texture; the splat will be mapped with this texture during rasterization and presents the visual property of its vector icon texture.

The vector glyph accounts for the final image rendered from the flow. We explore four vector glyphs: stroked lines, stream lines, particles, and arrows. The first three are demonstrated as the insets in Figure 4.3 (a), (b) and (c). The last (arrows) are demonstrated in Figure 4.6. Figure 4.3 (a), (b) and (c) show the vectors in the core part of the test dummy dataset [CM93] are rendered with stroked line, stream line, and particle glyphs, respectively. Figure 4.3 (d) shows the application of multi-glyphic texture splats. The vectors with different tensorial values are mapped with different vector glyphs.



(a) (b)

(c) (d)

Figure 4.3: The dummy test tornado dataset [CM93]. The tornado core is rendered with the inset vector icon texture in (a)(b)(c), respectively. The full dataset is rendered with 3 different icon textures (strokes, lines, and particles) corresponding to different velocity magnitudes.

4.2.2 Foreshortening of the Vector Icons

To represent the vector direction more accurately, we must foreshorten the anisotropic vector icons inversely-proportional to their projected length on the x-y plane in eye space. In other words, long vector icons should be used when the flow is parallel to the viewing plane while short vector icons should be used otherwise. Two plausible solutions for this are that we either rotate the textured splat such that it is always parallel to the vector field direction or that we simply shorten the quadrilateral geometry in the projected vector field direction. Unfortunately, neither method works for multi-variate vector fields, where the scalar reconstruction kernel needs to be preserved. Crawfis and Max [CM93] built a table containing the vector icon textures with different lengths. The z component, v_z, of the vector direction in eye space is used to index into the table. This again, required several repeated textures and wasted a large amount of texture memory. Our alternative is to shrink the vector texture by changing the texture mapping coordinates. It foreshortens the vector icon length by increasing its occurrence frequency. This foreshortening is view-dependent and hence, precludes any efficient strategy for calculating the texture coordinates and saving them in a display list or vertex buffer object. Fortunately, we can modify the texture mapping coordinates in the vertex program to implement this method. Since the vector icon texture is separate from the reconstruction kernel texture, we can treat their texture coordinates separately as well. When focusing on the flow direction visualization, a normalized vector direction is used to calculate the output texture coordinate. For the normalized vector direction, we interpolate the output texture coordinate, $output\_x_{tex1}$, on the x-axis from the following equation:

$$output\_x_{tex1} = \begin{cases} \min(1/d, f_{\max}) & if \quad input\_x_{tex1} = 1 \\ 0 & if \quad input\_x_{tex1} = 0 \end{cases} \tag{4.2}$$

Here, d is the projection length of the vector direction on the x-y plane and f_{max} is the maximum occurrence frequency of the icon. To prevent from producing an infinite number of vector icons when d approaches 0, the output x-coordinate is limited to f_{max} , or no more than f_{max} copies of the vector icon texture mapped to the same splat quadrilateral. There is no foreshortening for the vector icon if the projection length d is one, while the shortened vector icon repeats f_{max} times in the x-axis direction if d is equal to or smaller than $1/f_{max}$. According to our numerical experiments, the maximum occurrence of f_{max}=3 delivers a satisfactory visualization. We use normalized vectors to show the direction only, relying on other techniques to show the magnitude.

4.2.3 Dynamic Representation

Animation can be achieved by using a phase-shift through the overlapping vector icon textures [CM93]. Crawfis and Max [CM93] developed a set of textures and cycled through these textures to achieve the animation. Each texture was a cyclical shift of the vector icon in the negative flow direction represented by the texture. This presented problems in that all textured splats either had to move at the same speed, or a slight pause could be introduced to delay the texture movement and discrete jumps were made. These discrete jumps can be mitigated by using a larger set of textures, but at the cost of additional texture memory. Here, we have been able to reduce this to the use of only a single vector icon texture. To obtain the illusion of coherent motion, we design a vector

icon texture that is cyclically in the desired flow direction (we use the x-axis as the primary direction) and shift the texture coordinates in this direction for each time stamp. To reduce the hard edge between the overlap of the different splats, the splat texture is windowed. Crawfis and Max [CM93] used a larger Gaussian window to smoothly have the textures become transparent. This windowing was applied after they phase shifted the texture pattern and was pre-computed and stored with the set of textures. Applying this window to our phase shifted texture icons requires the opacity mask for the window to remain fixed on the textured quad. We only shift the vector icon texture coordinates along the desired direction in the vertex program. This requires separate textures, one for the reconstruction kernel, which also serves as the vector icon windowing mask, and one for the vector icon. These are then combined by the register combiners. This could not be achieved with OpenGL 1.1. Figure 4.4 shows a 2×2 tiling of the stroked vector icon texture. There is no seam between the adjacent textures. We consider an infinite texture created from these tilings, from which we roam through to produce the animation. The vector icon texture is shifted for each time stamp and then windowed while keeping the reconstruction texture untouched.

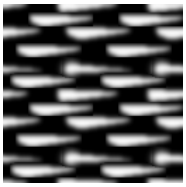


Figure 4.4: 2×2 periods of the vector icon textures.

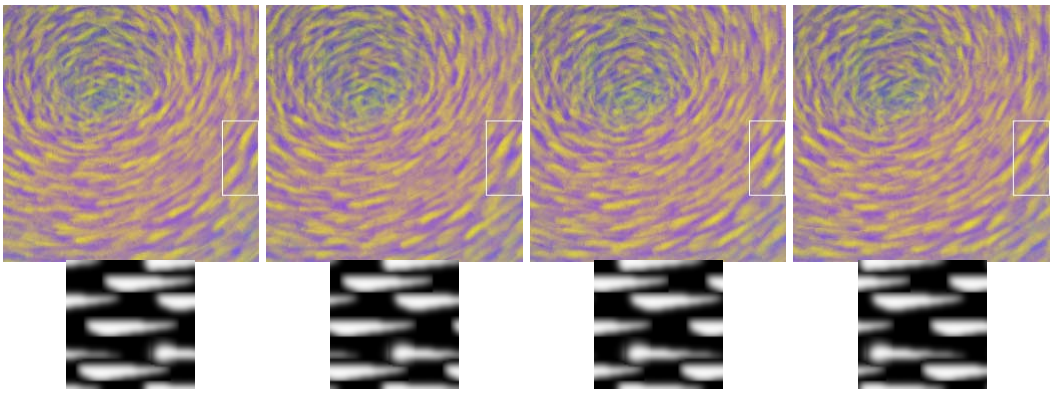


Figure 4.5: The close-up view of the vortex from figure 3(a). The framed regions in the top image show the shift of the vector icon on the tornado at the four successive time stamps. The bottom images show the corresponding shift of the vector icon texture.

Figure 4.5 shows the vortex part of the dummy test tornado dataset at four successive time steps in the four top images. The framed regions demonstrate the shift of the vector icons. The four bottom images are the shifted vector icon texture.

Unlike the static representation of the vector field like color or multi-glyphic texture, we map the velocity magnitude into the phase-shift step in the vertex program. The vector icons with different velocity magnitudes will move at their own speeds. This gives us a more intuitive understanding of the vector field.

4.3 Vertex Shader

To achieve volume rendering using 2D texture mapping hardware, we use a billboard technique [Lig] to orient splats, whenever the view direction changes, such that the splats always face toward the viewer. The OpenGL 2.0 vertex shader provides a vertex program to perform per-vertex operations, including the vertex transformation [LKM01,

Wyn]. To gain the interactive rendering rate, we need to avoid the overhead associated with per-splat computations on the CPU. The coordinates of the quadrilateral splat and the vector direction are first transformed into eye space. The rotation matrix in eye space is constructed to orient the textured icons aligned with the projected vector direction. The coordinates of the quadrilateral are then re-rotated to lie parallel to the screen. The pseudo code for the vertex program is as following:

1) Transform vector direction from object space into eye space;
2) Construct rotation matrix from the model view matrix;
3) Transform vertex from object space into the eye space;
4) Rotate splat from the rotation matrix in 2);
5) Perturb the starting position of texture mapping;
6) Index the proper vector glyph texture;
7) Calculate the shift for the vector icon;
8) Shift and foreshorten the x-coordinate of the vector icon texture.

We use a vertex shader program to orient the splat toward the viewer, rotate it to its projected vector direction, calculate the shifted icon texture coordinates, and index its vector glyph texture. All of these operations are performed on the GPU. This offers a significant improvement in rendering speed. In order to remove the artifacts due to the regular repetition of the vector icon texture, we randomly select an origin into the logically infinite tiled texture for each splat.

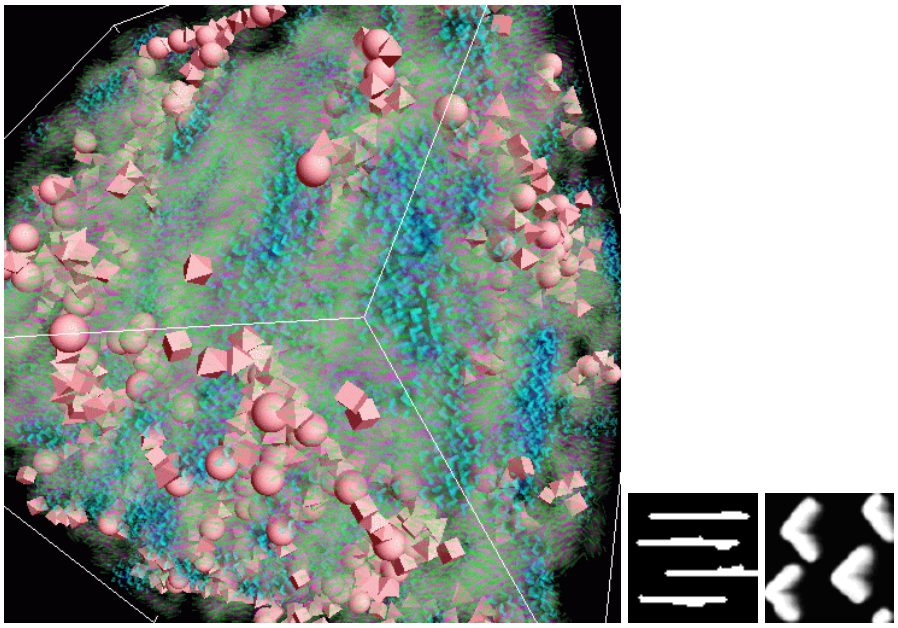


Figure 4.6: The image is rendered from aerogel dataset with two vector glyphs (lines, arrows). The vector field is coded not only by its color but also by the vector glyphs.

4.4 Experimental Results and Discussion

All performance results are generated on a Dell Precision 530 workstation equipped with 2.0 GHz Intel P4 and GeForce4 Ti 4600 with 128 MB video memory. Figure 4.6 shows the line and arrow vectors flowing through the volume rendering of the magnitude of an airflow through an aerogel (ultra light weight insulator) substance. Polygon data (sphere, cube, octahedron) representing the aerogel fibers are embedded into the volume rendering. The air flow velocity is coded by both color and the vector glyphs for static representation. In the animation, the velocity is also coded into the vector icon shift speed. Figure 4.7 shows a vector field of winds over the North America. The different

vector icons are selected to present different visual effects.

The rendering rates for the discussed datasets are listed in Table 4.1. From Table 4.1, today's consumer-level graphics hardware provides an acceptable FPS for interactive/dynamical flow volume rendering. More importantly, our method only uses one 64×64 texture for the scalar reconstruction kernel and one 64×64×m (m is the number of multi-glyphic vector icon textures). Both only use a single alpha channel component. Since the video memory is still a bottleneck for many applications, our vertex-shader-based texture splat shader reduces the large video memory usage as in [CM93]. Although modern graphics hardware has seen a tremendous increase in the available video memory, this reduction is still significant, freeing up video memory for other applications.

Dataset	**Size**	**FPS**
Tornado	96^3	8
Tornado	48^3	60
Aerogel	32^3	75
NA	128×64×16	12

Table 4.1: FPS for four vector field datasets.

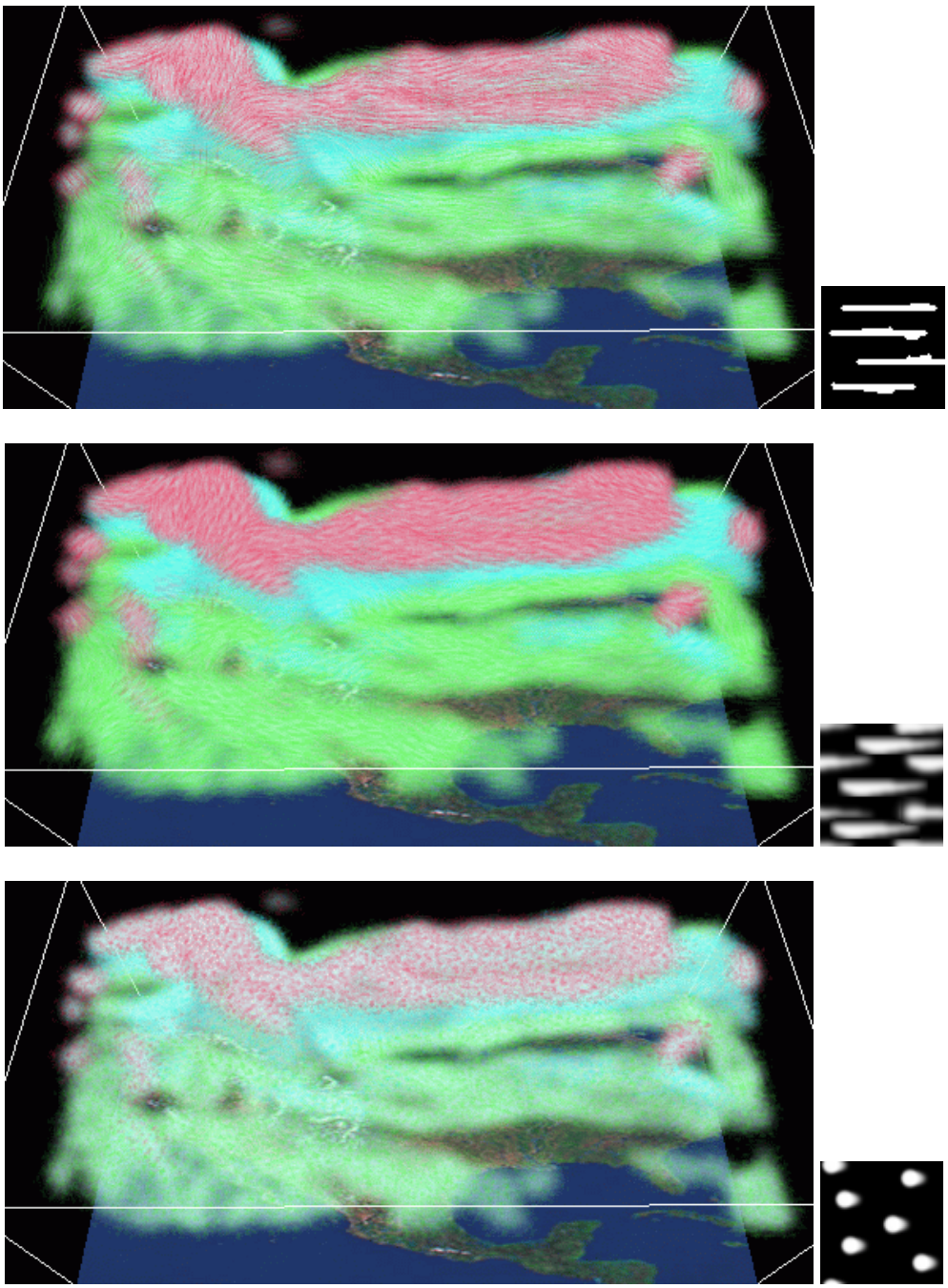


Figure 4.7: Wind on North America dataset. The left images are generated with their right texture icons, respectively. The velocity is coded by the vector icon color.

CHAPTER 5

IMPLICIT FLOW VOLUME SHADER

5.1 Introduction

Traditional flow volume rendering, as proposed by Max et al. [MBC93], constructs an explicit geometrical representation of the separating volume using a streamline advection operator applied to the underlying vector field. The geometry is rendered using an unstructured rendering technique. Information within the flow volume boundary is usually incorrect unless a detailed refinement of the interior volume is specified. Van Wijk [vW93] extracts implicit stream surfaces from a three-dimensional vector field which are rendered using traditional polygonal rendering methods. Although the flow structure can be well-tracked by the stream surfaces, the flow information inside or behind the surface is not visible. Multiple stream surfaces could be generated, but these would either occlude each other or a polygonal rendering system that can accurately support semi-transparent rendering of surfaces would be needed. We examine the problem of extending the implicit stream surfaces to that of implicit flow volumes. We develop a new rendering shader of the implicit flow. By careful tracking and encoding of the advection parameters into a three-dimensional texture, we achieve high appearance

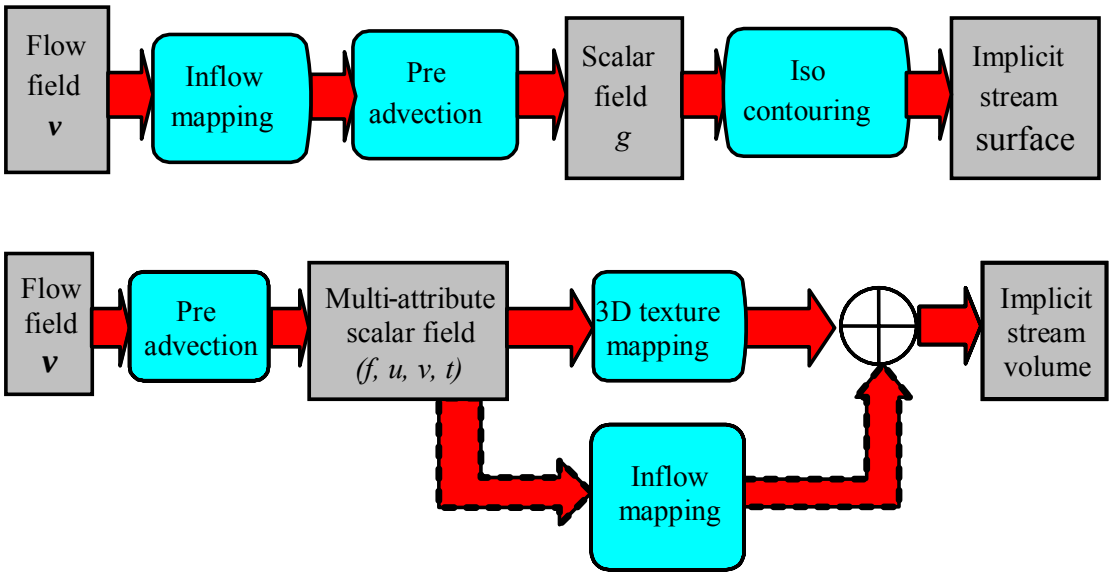


Figure 5.1: Visualization diagrams for van Wijk's implicit stream surfaces (top), and our implicit flow volumes (bottom).

control and flow representation with real time rendering.

Our overall goals in this study include the examination of more dynamic and detailed flow visualizations in three-dimensions, in particular, as it relates to advection-based flow volumes. Our criteria, thus includes real-time interaction with the advection operation, support for animation through the flow, and visualization of large areas of the flow with minimal clutter.

5.2 Related Work

Various techniques have been proposed to render vector fields. We focus on the work for rendering 3D vector fields. Crawfis and Max [CM93] introduce a textured splat method to provide a dense global visualization of the 3D vector field. Line Integral Convolution,

LIC [CL93], provides another dense visualization with more accurate local features. Rezk-Salama et al. [RHTE99] explore rendering volumetric LIC using 3D texture mapping hardware to examine the flow fields. Auxiliary clipping geometries are used to reveal the LIC pattern. Another dense visualization technique is the Image-Based Flow Visualization (IBFV) technique [vW01, LJH03]. Telea and van Wijk [TvW03] extend the IBFV method to 3D IBFV to visualize three-dimensional flow fields. These methods provide fine-grain localization of the flow, with the aim of texture synthesis for a more global perception of the vector fields. Avoiding excessive clutter through tuning the various parameters, or restricting the range of the visualization can be difficult with these systems.

In contrast to the above dense visualization techniques, Zöckler et al. [ZSH96] use illuminated streamlines to depict the 3D vector field. Other geometry-based methods include the stream polyhedron [SVL91], stream surfaces [Hul92], implicit stream surfaces [vW93], flow volumes [MBC93], streamballs [BHR*94] and saddle connectors [TWHS03]. Li et al. [LBS03, SLB04] propose a hybrid method, where geometry is first constructed and voxelized into a coarse mesh. Each voxel in this coarse mesh is replaced with a dense 3D volume texture. The voxelized geometry, streamlines in their case, is fixed during the rendering and cannot be changed interactively.

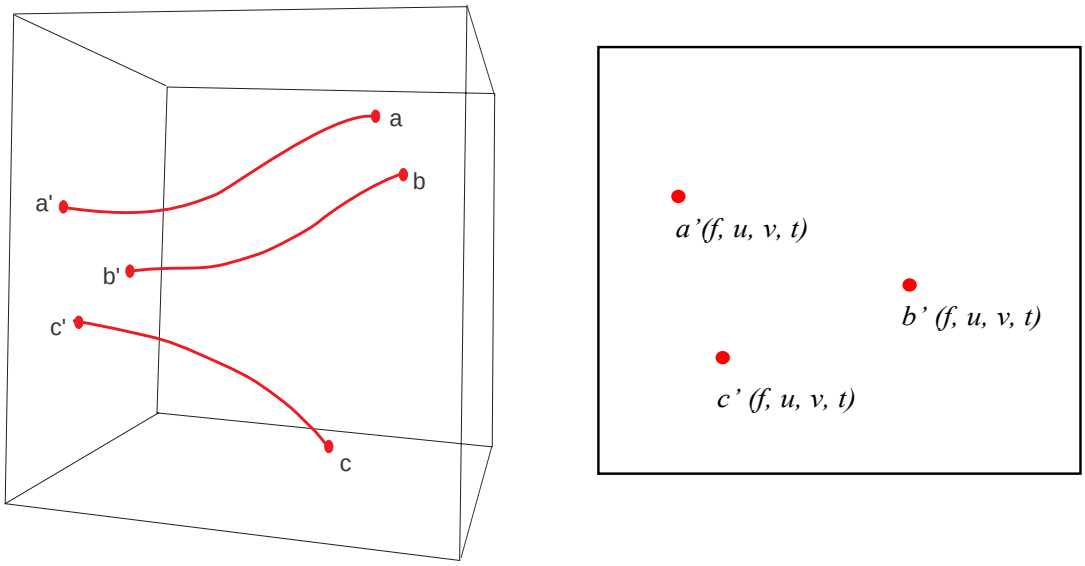


Figure 5.2: Backwards advection. Left: Three points, a, b and c, and their streamlines from the termination face. Right: each point (streamline) is assigned a 4-tuple, *(f,u,v,t)*, according to its advected (backwards) position on the termination face.

5.3 Functional Mapping and Implicit Flows

Given a flow field (vector field), we first determine a multi-variate field in which each sample point in the field is assigned attributes from the flow. The basis of our implicit flow volumes extends from the implicit surface definition of van Wijk [vW93]. The visualization diagrams for van Wijk's implicit stream surface and our implicit flow volume are illustrated in Figure 5.1. He associates a scalar field with the inflow boundary of the computational grid, and then for each remaining grid point, he traces a streamline backwards in the flow until it reaches the boundary (ignoring critical points within the flow). The scalar field is evaluated at this location, and the grid point is assigned this

scalar value. This amounts to a mapping of the 3D vector field to a 3D scalar field, $\mathscr{R}^3 \rightarrow \mathscr{R}$. An iso-contour surface is then extracted from this resulting scalar field to provide the stream surface. In addition to the obvious volume versus surface difference[1], three major differences exist between our technique and that of van Wijk. First, we either delay the specification of the scalar field on the inflow boundary, or eliminate the mapping onto a scalar field entirely. This allows us to develop new flow volumes without having to recompute the costly advection operations. Secondly, we associate several additional attributes with each sample point which allow for better user interaction, complex feature specification and enhanced surface representations. Finally, we allow for the user specification of many arbitrary boundary surfaces, which we call *termination surfaces*, indicating the termination of the backwards advection process. These can be used to place a termination surface around each critical point, allow for the specification of inflow and outflow boundaries [MBS*04] or as a user-controlled segmentation of the flow. Westermann et al. [WJE00] also use an implicit method to convert the vector field to the scalar field by storing the advection time. They render time surfaces using a level-set method by taking advantage of 3D texture mapping hardware. Their method is pretty similar to van Wijk's implicit method, but without the need for the inflow mapping.

A general functional mapping is associated with each sample point. Here we define a sample point as any location in three-dimensional space, preserving a continuous mapping operation. In practice, we will generally associate a sample point with each vertex in either the underlying computation grid or a superimposed voxel grid. There are many attributes that can be derived or mapped onto each sample point. Local operations,

[1] Van Wijk actually points out the extension of implicit stream surfaces to stream volumes, as well as a flow of ink metaphor.

such as velocity magnitude, vorticity, etc. provide simple filters. For implicit flows, we associate, at a minimum, a termination surface ID indicating which surface the backward streamline intersected first, the coordinates on the termination surface in a local coordinate frame to the surface, as well as the advection time required for the flow to reach the termination surface (backwards, or conversely, the time required for a point on the boundary to reach the sample point). This is illustrated in Figure 5.2. Additional attributes, such as the maximum velocity magnitude along the streamline, average density along the streamline, etc., can also be calculated and stored in this preprocessing stage. Thus, in general, we have an operation computing a mapping from $\mathscr{R}^3 \rightarrow \mathscr{R}^n$. The focus in this chapter will be restricted to maintaining the four attributes mentioned above: termination surface ID, parametric position on the surface, and the advection time. Hence, for each sample point we store a 4-tuple, *(f,u,v,t),* containing these values. This 4-tuple representation will be the basis for all of our future renderings in this chapter.

5.4 Rendering of Flow Volumes

Our task now is to examine methods for either rendering such a field or extracting more meaningful regions from this space. This suggests another mapping, one from the attribute space to optical properties for rendering. In the sections that follow, we will define a few such mappings. A primary criterion for such a mapping rests in providing flexible and robust mappings that provide an intuitive and simple interface. In order to better explore the flow, the user needs to be able to interactively adjust and control this mapping. This also suggests that a graphics hardware shader be suitable for such flow

manipulation and rendering, considering the high performance and the high programmability of today's graphics hadware.

In this chapter, we present a 3D texture mapping volume shader to model and render the implicit flow volumes. This technique renders the implicit 4-tuple flow field directly without the inflow mapping to a scalar field, taking advantage of modern graphics hardware. With the support of the dependent texture as the inflow mapping, we can change the appearance and representation of the 3D flow volume using advanced volume shaders. The advantages of this rendering method are high interactivity and fine texture details rendered throughout the 3D flow volume.

5.5 3D Texture Mapping Volume Shader

Traditional three-dimensional texture-based volume rendering takes as input a pre-shaded RGBA voxel grid. This is loaded into three-dimensional texture-memory and image-aligned proxy geometry is rasterized with three-dimensional texture coordinates specified, such that an interpolation of the texture-map values is painted across the proxy geometry. This set of proxy geometry is rendered in a back-to-front (or front-to-back) order, compositing the next slice over the partially computed image. Recent research [WE98, MMC99, KKH01] illustrate the benefits of using post-classification. Here, the original scalar field is mapped into the three-dimensional texture memory. The proxy geometry is then used to interpolate a slice of the underlying scalar field. Each interpolated value on this slice, then needs to be mapped to an appropriate RGBA value for compositing. This is supported through dependent textures in most modern graphics hardware.

Dependent textures allow both the representation and appearance of the 3D

volume to change. Li et al. [LBS03] use small three-dimensional dependent textures, indexed using a trace volume generated from voxelized streamlines. The dependent texture allows for colorful volumetric textures along the streamlines, and due to its small size is more easily replaced, allowing for animation effects along the streamlines. We extend their concept of a trace volume to an implicit flow volume in which each voxel is an *n*-tuple as defined in section 5.3. For interactive user-controlled exploration, the system must support re-painting the dependent texture in real time. In order to accomplish this, the number of texels being updated in the dependent texture needs to be limited. Our underlying functional mapping is a 4-tuple mapped to an RGBA normalized format. OpenGL supports up to four texture coordinates, but does not support four-dimensional textures, so a 4-component dependent texture is only theoretically possible. Besides, changing every value in a 4-dimensional texture becomes prohibitively expensive as the resolution of the dependent texture grows. Three-dimensional dependent textures are supported, but if our goal is to allow the user to control the appearance of the flow throughout the entire volume, a dependent texture of at least the same size as the underlying voxel grid would be required. This differs from Li et al. [LBS03], in that our goal is to change the underlying trace volume dynamically. Updating the entire volume in real-time is not feasible for large volumes. This also would greatly reduce the amount of texture memory available for the implicit flow volume.

Our focus instead, has been on reducing the mapping down to a 2D parameterization of a single termination surface. Our approach, allows the user to paint the dependent texture colors and opacities directly on this surface [HH90]. We call this

dependent texture, the *inflow texture* in the subsequent sections, as it dictates the paint that is carried from the termination surface into the flow. The next few sections provide details on a few of these choices, as well as additional techniques which extend this parameterization to utilize more attributes from the underlying 4-tuple in the implicit flow representation.

Figure 5.3 shows the general diagram for a volume shader to render the implicit flow volume using inflow texture. The implicit flow field is loaded into a 3D texture and the inflow texture is loaded into a dependent texture as look-up table to produce different flow representation and visual effects.

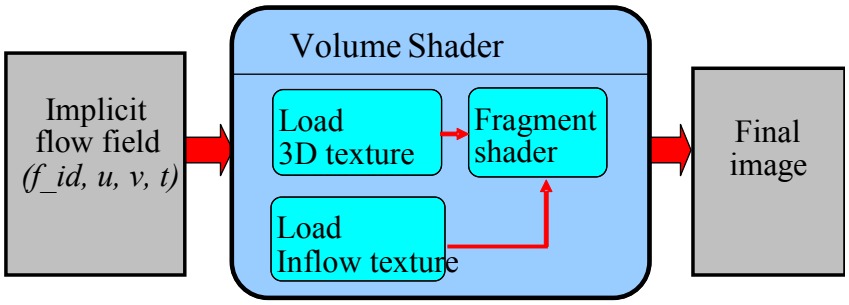


Figure 5.3: The volume shader with inflow texture to render implicit volume.

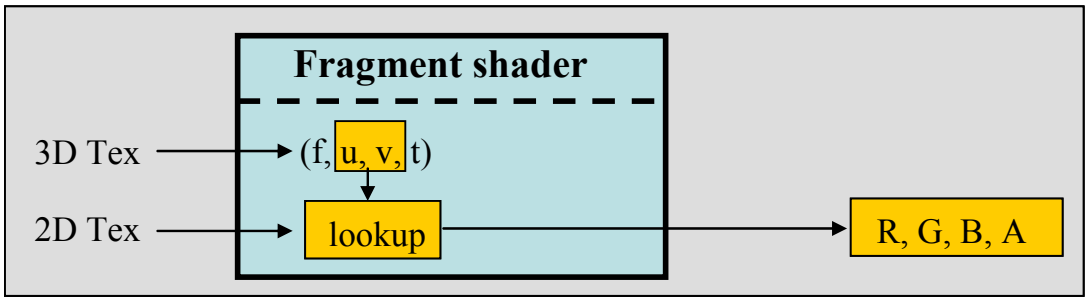


Figure 5.4: Volume shader for inflow mapping Φ(f,u,v,t) = 2D Texture (color + opacity) .

5.5.1 User-Controlled Painting

Without loss of generality, we consider an implicit flow volume which only one termination surface exists, i.e., the backward advections of all sampling points terminate on this surface. The simplest such surface would be the bounding box for the flow field. A dependent texture mapped to this box is used as our lookup table. The *(u, v)* in the 4-tuple, *(f, u, v, t)*, is employed to index into the dependent texture, producing the current fragment's color and opacity. By changing the alpha mask, we can dynamically change the three-dimensional representation of the flow. Figure 5.4 shows the volume shader for inflow mapping *Φ(f,u,v,t)* defined by a 2D Texture (color + opacity). The implicit flow field is loaded into the 3D texture and the inflow mapping is loaded into the 2D dependent texture in Figure 5.4.

With our user interface, we can brush the inflow texture to get arbitrary representations of the flow. Figure 5.5 (top) shows an image in which the user hand-painted *vis 2004* on the inflow texture on one face of a bounding box. In addition to hand

painting, the user can import any image for use as the inflow texture. Figure 5.5 (bottom) has the IEEE Visualization 2004 conference logo used as an opacity and color texture. The painting modes are supported for adding paint to the inflow texture. The previous texture can be cleared and new paint added from the user's current brush, providing a moving flow volume. The previous texture can simply be added to, building out regions of interest in the flow. An eraser (a brush that reduces the opacity) is also supported for refining these regions of interest. Finally, the previous texture can be *faded* out over time by first reducing its opacity and then adding new paint under the user's control. This provides a motion blur of the flow volume as it moves through the field.

More complex termination surfaces can easily be supported, provided a parameterization exists. This is in general, a hard problem. In addition to flat planes, we currently support spherical and cylindrical termination surfaces (useful for bounding a neighborhood of a source), and a rectangular box termination surface. The parameterization of the box is supported through the use of OpenGL's cube-maps. This allows for six independent termination surfaces, onto which the user can paint an inflow texture. Figure 5.6 shows two images generated with a cube map for inflow texture on the tornado dataset. By drawing spots on the cube map, we can easily change the flow representation of the dataset.

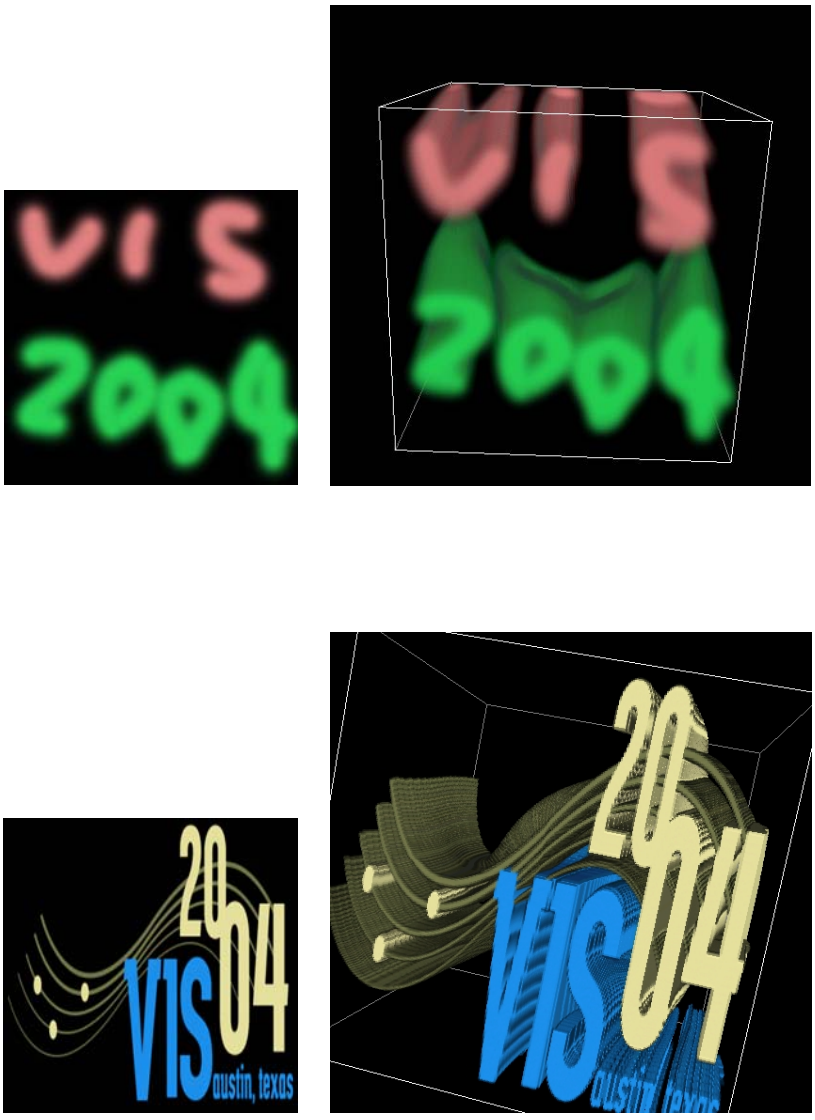


Figure 5.5: Two different inflow textures advected thru a same flow volume.

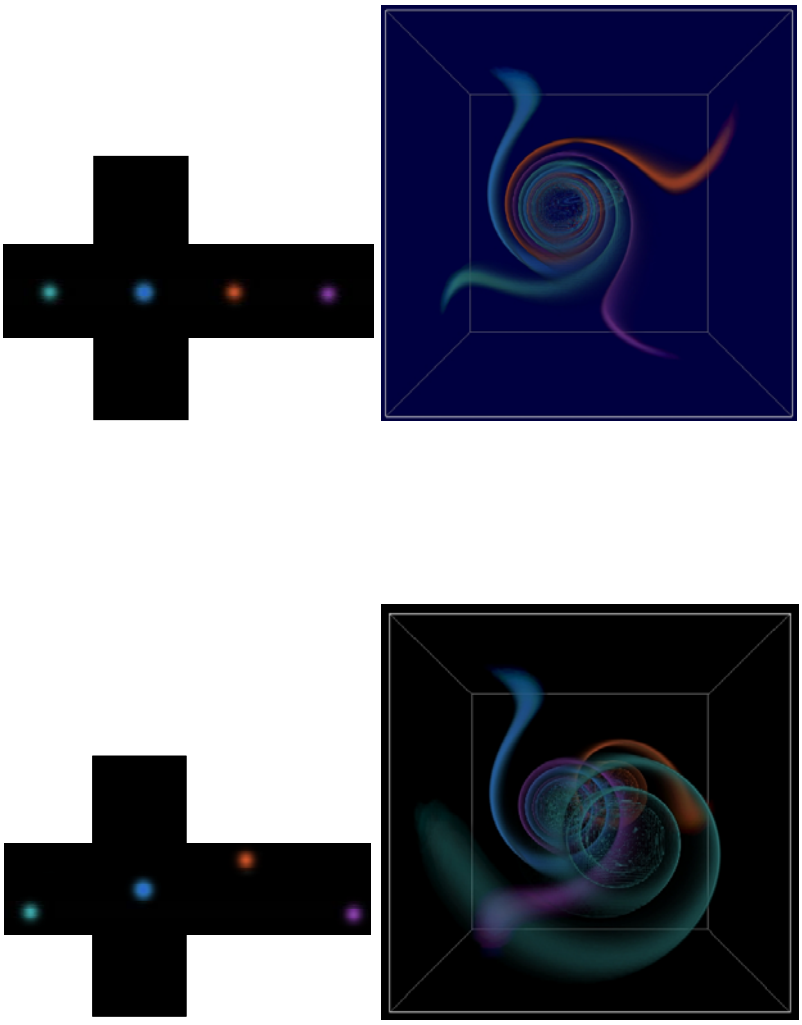


Figure 5.6: Inflow mapping on a cube map texture ((left column)) for tornado dataset.

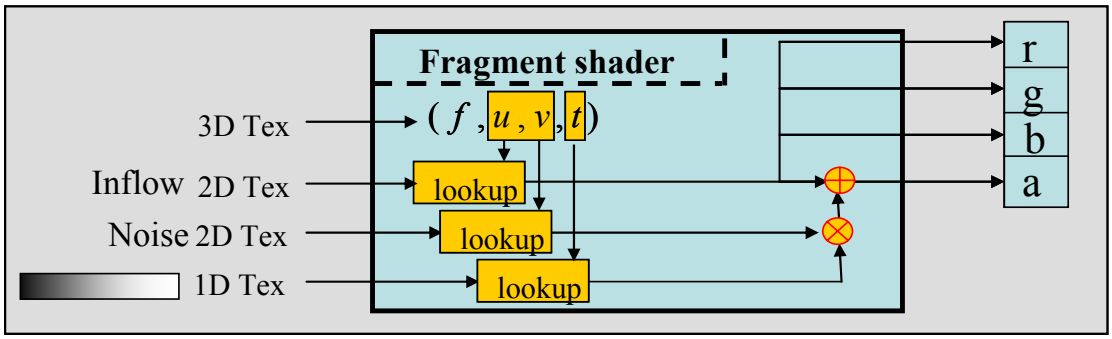


Figure 5.7: Volume shader for dual inflow mapping and animation.

5.5.2 Dual Inflow Textures

Periodic dye injection can help understand the interior structure and highlight local features in the flow. Shen et al. [SJM96] use a "smeared" noise texture to simulate dye. Instead, we use a multi-texture technique. In addition to the user defined inflow texture, we create a separate dual-inflow texture. The support for dual inflow textures, allows for a separate high-frequency texture without requiring the user to painstakingly paint in such details. A high-resolution dependent texture can be used for this purpose. This does not require a large amount of texture space, provided the high-frequency texture is periodic and tile-able. Figure 5.7 shows the volume shader diagram for dual inflow texture mapping. An additional 1D texture in the diagram is attached for flow animation which will be discussed in the next section. Figure 5.8 (left) embeds a dual inflow texture with a regular grid pattern and a Poisson disc pattern. By changing the frequencies of the noise and grid texture, the flow is visualized to assume different details inside the volume as in Figure 5.8 (middle and right).

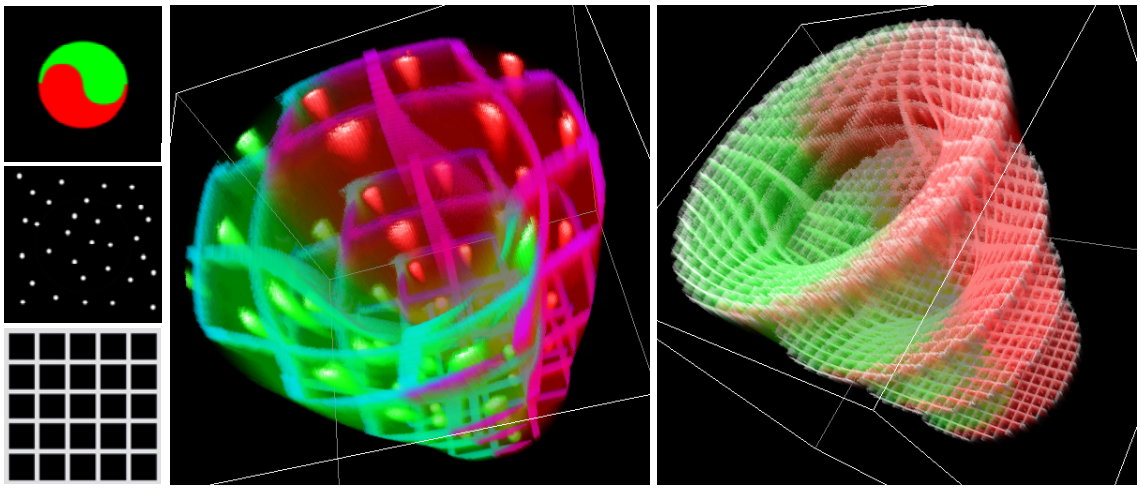
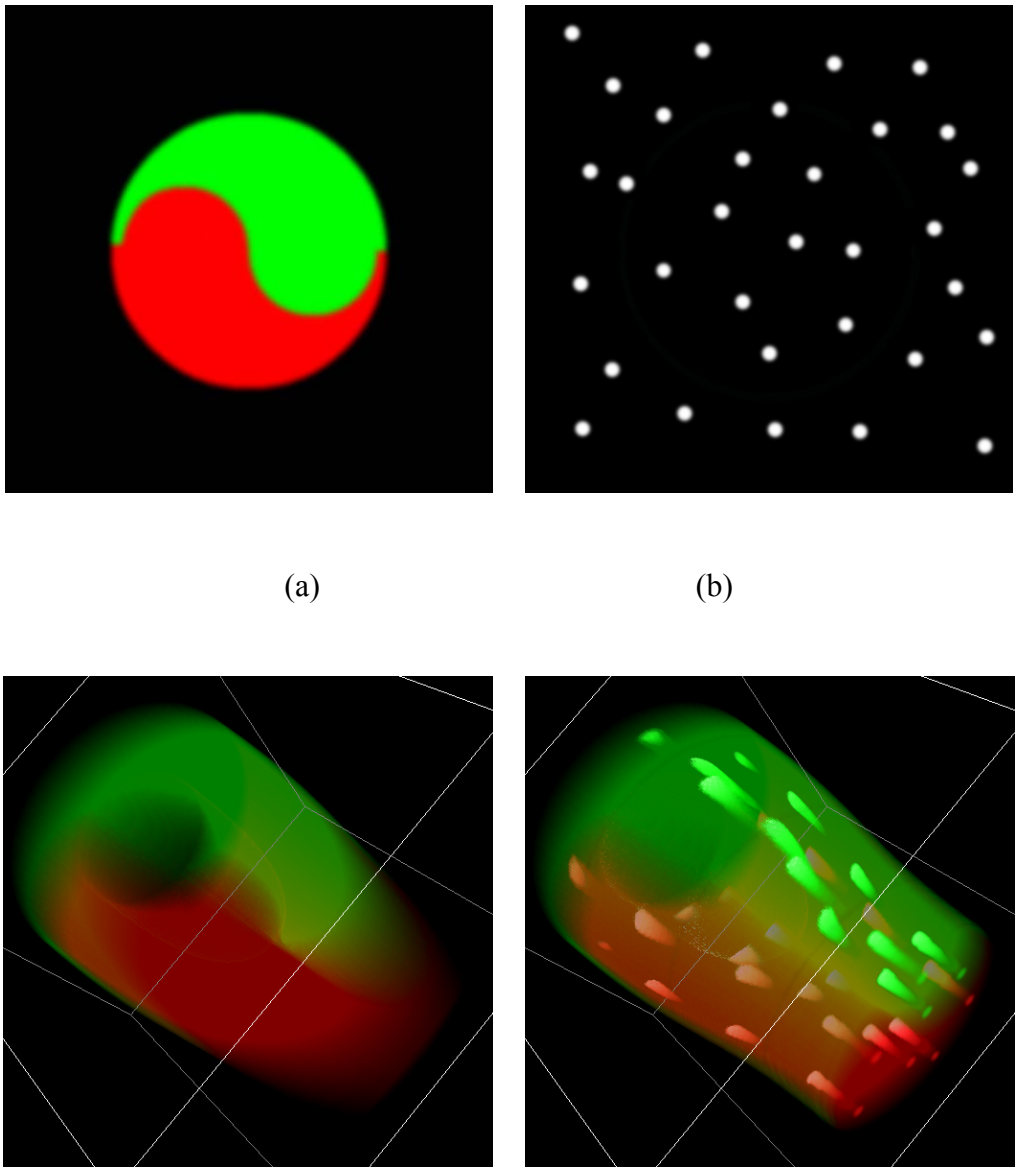


Figure 5.8: Complex cross-section for the inflow with dual-texture support.

5.5.3 Inflow Texture Animation

Max, Becker and Crawfis [MBC93] included a simple modulation of the opacity as a function of the advection time for their flow volumes. By phase shifting this modulation function, they were able to animate smoke puffs along their flow volumes. In our implicit flow volume representation, the advection time has been encoded into the 4-tuple for each sampling point. We define a one-dimensional opacity table containing an opacity modulation. For animation, we phase-shift this opacity modulation for each time step. The advection time information of the sample point is used to index into the opacity table. This is combined with the dependent textures for the inflow texture, using multi-textures. This produces a puff-like motion in the flow. By adding another dependent texture, we can also encode the *age* of the paint on the inflow texture. Adding this age to the

advection time releases the paint from the inflow texture through the flow field (see the shader diagram in Figure 5.7). We can also automate this to provide flow representations using particles, animated streamlines and propagating time fronts. Each inflow texture can be animated separately. This allows one to model a periodic dripping or injection of colored dye into the flow volume. The underlying flow volume retains its global characteristics and multi-colored sub-flows are passed through the flow volume. For the images in Figure 5.9, an initial inflow texture, as shown in Figure 5.9a and a dual inflow texture as shown in Figure 5.9b were used to generate the image in Figure 5.9d. The image in Figure 5.9c shows the flow volume without a high-frequency detail texture. Here, two dependent textures are used, both indexed similarly, with a multi-texturing operation that replaces the paint from the inflow texture with the dual inflow texture.



(a) (b)

(c) (d)

Figure 5.9: a) The inflow texture specified by the user. b) A particle distribution. c) The result from the inflow texture only. d) The result obtained by combing the inflow texture and texture b).

5.6 Experimental Results and Discussion

In this chapter, we have described an inflow texture based volume shader for implicit flow volume rendering. In this section, we compare the advantages and disadvantages of this technique with respect to explicit flow volumes [MBC93]. Table 5.1 summarizes our comparison between these two techniques.

The implicit flow is generated by pre-advecting the flow field and storing the advection information for each voxel in the implicit flow. When we subsequently construct a stream volume, no integrator is required to compute streamlines through the flow field. Since this is a pre-computation, care can be taken to ensure accurate streamline advection. We use an adaptive fourth-order Runga-Kutta algorithm. The 3D texture mapping method renders stream volumes using a dependent texture. Explicit flow volumes are constructed using an advection algorithm during run-time. The 3D texture mapping has an advantage when the inflow boundary is changed, as it does not require any re-computation. The other technique needs to re-compute their flow volumes through advection, which can be a costly operation. In our experiments, the 3D texture mapping can achieve roughly 10 FPS for a 128^3 implicit flow dataset. All experiments are performed on a PC with a QuadroFX 3000 graphics hardware and a Pentium IV 3.2 GHz processor. Although, it should be pointed out that our technique runs fairly interactive for the datasets we have tested, a true performance comparison is not provided, due to the many parameters each technique requires for the specification. For any given technique, we can find a case where it would be the fastest, or the slowest. Nevertheless, there is a key differentiating factor that we wish to highlight among these flow techniques.

5.6.1 Volumetric Details

In the traditional method, a cross section is specified by a low-resolution polygon. The quality of the cross section and the flow boundary is limited by user specification. Typically, the quality is poor. Furthermore, the distribution of any optical properties across the cross section is ill-specified. In order to allow for changes of the optical properties across the initial smoke generator, a subdivision of the cross section (and hence the resulting explicit flow volume) is required. Most explicit flow volume renderings utilize a constant color and extinction coefficient.

For the implicit methods, the cross section is specified using a general inflow texture. The complexity of the cross section and the resulting flow boundary is thus determined by the resolution of the dependent texture for the 3D texture mapping technique. An extremely high virtual resolution is possible with the dependent textures. No assumptions about the underlying volume rendering model are made in our system. In fact, an arbitrary fragment program can be used to compute the volume rendering. This allows for volumetric straw textures, etc.

The implicit stream volume includes flow properties at all sampling points within the flow. This flow detail information is stored in the implicit flow volume representation and can be used to modify the color and opacity in a 3D texture based graphics hardware rendering shader. Such shader can easily provide the representation and appearance control of the flow (see Figure 5.5, Figure 5.6, and the images in [LBS03]).

5.7 Conclusions

We have developed a set of graphics hardware shader for implicit flow volume rendering. The inflow textures are explored in these shaders to control the flow appearance, change the flow representation, and add the volumetric details inside the volume. We compare our technique with the traditional explicit flow volume rendering on many aspects (summarized in Table 5.1) and highlight the most important feature of our 3D texture based implicit flow volume shaders. Our experimental results show that our graphics hardware based shaders are very suitable for visualizing the implicit flow with high performance and on-the-fly control on flow representation and appearance.

	Traditional Flow Volume	Implicit Stream Volume Shader
Requires pre-processing	No	Yes
Advection	Advection during the volume construction	Pre-advection
Flow volume construction	Through advection	Using dependent textures
Representation	Explicit	Implicit
Initial Starting Location	Anywhere	User-defined Termination surfaces (pre-computed)
Stream surface / time surface	Easily added	No
Rasterization / rendering range	Render only the flow area	Rasterize the entire volume
Requires recomputation	Yes	No
Non-regular grids	Easily supported	Requires voxelization
Cross section specification	Polygon	Per-pixel mask function
Cross section quality	Limited by user specification, typically poor	Resolution of the dependent texture
Boundary quality	Dependent on polygon	Dependent on dependent texture
Details / correctness	Without mesh refinement, misses details in flow.	More accurate
Rendering performance	Dependent on the number of tetrahedra	Dependent on voxel grid size
Volume size	Arbitrarily large volume	Limited by the texture memory of the display card
Source, sink critical points	Fine	Requires critical point detection

Table 5.1: Implicit flow volume shader vs. traditional flow volume rendering technique.

CHAPTER 6

INDIRECT SHADER SYNTHESIZER

6.1 Introduction

A *shader* in computer graphics normally indicates the rendering program to produce a 3D image from volumetric data and/or surface geometries. The shader can be implemented to run on both CPU and GPU. The modern graphics hardware provides the high programmability on GPU [ATI, NVIDIAa, NVIDIAb]. The GPU-based shader has been becoming more and more efficient, flexible, and popular.

Generally speaking, in a rendering pipeline, there are two kinds of shaders: vertex shaders and pixel shaders (or fragment shaders in OpenGL) [Mic02, SA04]. The vertex shader is responsible for transforming the input vertex into clip space; the pixel shader is responsible for processing the pixels from rasterization and outputting the color into the frame buffer. Most recently, a geometry shader is introduced into the latest graphics hardware [NVIDIAb] to generate new geometry primitives from those that were sent to the beginning of the graphics pipeline. We are more interested in using a shader program to produce the color for a pixel in the final image. Hence, in this chapter, we limit our discussion to the pixel shader and the term “shader” refers to a pixel shader if not

specified.

In a single shader application, all pixels in the output image are created via the same shader and rendering setup (light, texture, camera, etc). In this chapter, we present a shader synthesizer to combine different shader rendering effects to create a highly informative visualization of the input data. The idea for shader synthesizer is that the different pixels may be computed with different shaders and the final color of the pixel is the combination of the multi-shaders.

6.2 Related Work

In this section we will focus on previous work about multi-shader rendering and the synthesis between the different shading results.

McGuire [McG05] describes a shader framework called "SuperShader" that renders many effects on surfaces. It allows arbitrary combinations of the rendering effects to be applied to surfaces simultaneously. The effect shaders are generated and optimized at runtime. One key problem in multi-shader rendering system is to manage the rendering order of the shaders (a.k.a permutation problem). McGuire [McG05] solves this problem by generating various shader source codes from source code snippets. Hargreaves [Har05] shows how to automatically expand large numbers of shader permutations from a smaller set of input shader fragments. McGuire et al. [MSPK06] present an "abstract shader tree" system for generating complex GPU shaders through automatic combination of primitive shading functions. In [MDTP*04], McCool demonstrate an approach to connect and combine shader programs using algebraic operators. In a more recent work [TD07],

Trapp and Döllner transform the shader source code fragment into an intermediate representation which is associated with the predefined semantics for combination at run time.

In visualization community, many multi-shader rendering work had been done in the field of focus+context rendering and importance-driven rendering. Kruger et al. [KSR06] developed an interactive context preserving volume rendering to achieve focus+context rendering. They use multi-shaders to depict the different body parts like skin and bone. The final color is a weighted average of all shaders involved. Viola et al. [VKG04] propose an importance-driven volume rendering. The voxel in the data is assigned an *object importance* which encodes a visibility priority. This property determines whether a more-important region is behind a less-important region. When this occurs, the less-important-region will be rendered with a reduced opacity. Thus the objects of interest are always clearly visible. They use Maximum Importance Projection (MImP) to determine the most important object location for each ray and the less-importance objects in the front are removed or become more transparent to achieve a cut-away view. Most recently, Plate et al. [PHF07] developed a multi-volume shader framework to render the intersection between the datasets with different resolution. They define a set of convex polyhedral volume lenses associated with one or more volumetric datasets. The lenses can be interactively moved around while the region inside each lens is rendered using interactively defined multi-volume shaders. Their result shows the very promising shading behavior of the combination of the resultant imagery from multi-shaders.

Texture image synthesis has been well-studied in the past few years. We will focus the related work on the stitch or blend between multiple image patch on the arbitrary surface. Praun et al. [PFH00] proposed a *lapped texture* technique for surface texture synthesis in 2000. They used an irregular texture patch and iteratively pasted it onto the surface. The placement of the patches is oriented according to a pre-defined vector field on the surface. An alpha-mask for the patch is created for further alpha-blending to smooth the transition between the overlapped patches. Their approach can create very nice result. However, the method is sensitive to input textures and is unsuitable for textures with strong low-frequency components, boundary mismatches, or a singularity point. Most recently, Takayama et al. [TOII08] extend this work into "lapped solid texture". They classify the solid texture according to its tilability/anisotropy and a tensor field is created to match this tilability to guide the solid texture fill-in orientation inside the mesh. Similarly to 2D lapped texture, they created a 3D alpha mask for alpha blending between the lapped solid textures.

6.3 Shader As Function Mapping

We generalize a shader as a function mapping:

$$\Phi : \mathbf{x} \times \mathbf{v} \rightarrow \mathbf{c} \tag{6.1}$$

where $\mathbf{x}$ is the 3-tuple of the input pixel position, $\mathbf{v}$ is a vector of attributes associated with each pixel, and $\mathbf{c}$ is the output color (RGBA) for the input pixel. The attributes in equation (6.1) include, but are not limited to, normal, texture, texture coordinates, and view direction. These attributes are used in the shader program to compute the output pixel color $\mathbf{c}$.

We apply the unary or binary operations between one or more shaders attached to the same pixel. In our study, we explore the following operators:

- ***Complement***. The final color is the complement of the shader output.
- ***Min***. The final color is the component-wise minimum between the outputs of the two shaders.
- ***Max***. The final color is the component-wise maximum between the outputs of the two shaders.
- ***Over***. The final color is the blending composition between the outputs of the two shaders.
- ***Replace***. One shader output replaces the other shader output as the final output.
- ***Weight***. The two shader outputs are weighted and added together.
- ***Sum***. The final color is the addition (with clamp) of the two input shader outputs. The ***Sum*** operator is the special case of ***Weight*** with the two weight coefficients equal to one for both shaders.

The outputs from the above operations are normalized (clamp) to fit the rendering pipeline. In addition, the shader operations can be cascaded.

6.4 Shader Classification

A pixel shader can be an arbitrary function mapping to generate the output color for any input pixel. We classify the shaders into several types such that we can use a different user interface to handle the synthesis between the shaders. Our classification is based on the functionality of the shaders. These shaders include:

- Null shader
- Photorealistic shader
- NPR shader

- Procedural shader
- Volume rendering shader

6.4.1 Null Shader

A null shader indicates no shading occurs for the input pixels. The shader ID of 0 is reserved for the null shader in our shader synthesizer system. The null shader is especially useful to produce an erosion-like object [PH89].

6.4.2 Photorealistic Shader

The photorealistic shader is most commonly used in graphics applications. The output color for each pixel is computed based on local shading models [Pho75, BN76] or global illumination [PH04, DBB06]. Texture mapping can be applied in the shader program. In most cases, this shader tries to generate a photo-realistic effect image.

6.4.3 NPR Shader

Non-photorealistic rendering (NPR) focuses on enabling a wide variety of expressive styles to convey the most important information in the output image. In contrast to a photorealistic shader, which focuses on photorealism, NPR is inspired by artistic styles such as drawing, painting, technical illustration, and animated cartoons [DS00, KMT*97, LME*02].

6.4.4 Procedural Shader

In a procedural shader, the pixel color is calculated by the procedural function such as the well-known Perlin's noise, turbulence functions [Per85, PH89]. The procedural shader is

very suitable for modeling and visualizing the realistic texturing of complex surfaces, especially simulating a sculpted appearance of objects. Marble color, wood ring, and the gaseous phenomenon can be generated from such procedural functions [EMP*03, KCR99].

6.4.5 Volume Rendering Shader

A volume rendering shader creates the image directly from volumetric data without generating the geometry. In volume rendering, by changing the blending mode to composite the samples along the ray casting into the volume, we can create an image with distinguished appearance to convey the different information of the data. Commonly used shaders include direct volume rendering (DVR), X-Ray, and maximum intensity projection (MIP) [Lev90, XC03, DCH88]. A transfer function can be applied to perform post-classification [EKE01] in DVR.

6.5 Shader Design

We investigate some typical procedural, NPR, and volume shaders used in our study.

6.5.1 Granite Shader

Granite is defined via the well-known Perlin noise [Per85, PH89]. In our study, we define the granite as in equation 6.2:

$$Granite(\mathbf{x}) = \min(intensity * Noise(\mathbf{x} * frequency), 1.0) \qquad (6.2)$$

where *frequency* determines the number of occurrences and the spot size in the granite; *intensity* determines the granite gray intensity. The above equation returns the granite

color at the given position. Figure 6.1 (left) shows the granite appearance on a Klein bottle [Ros06].

6.5.2 Toon Shader

Toon shading is a technique to render objects in a "cartoon-style". The silhouette of the geometry is enhanced and normally rendered in black. In addition, the smooth lighting values are calculated for each pixel and then mapped to a small number of discrete shades to produce the characteristic flat look (see Figure 6.1 right [Ros06]).

6.5.3 Layered X-Ray Shader

An X-ray image is very suited for visualizing high density materials, such as bone in medical datasets. However, depth information is undefined in the classical X-Ray integration. We can create the X-Ray image layers for different image-aligned slabs in the volumetric data. These image layers can later be blended together to generate a new layered X-Ray image with more depth cue information (see Figure 6.10 Top).

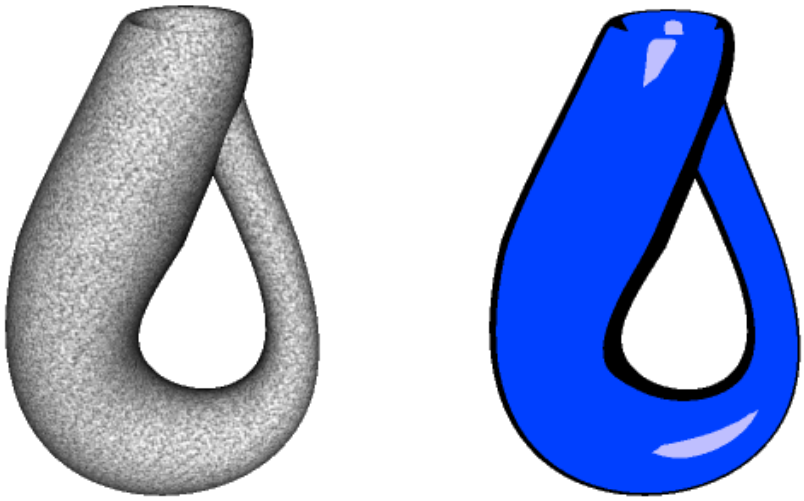


Figure 6.1: The granite and toon shading on a Klein bottle. Courtesy of 3D Labs [Ros06].

6.6 Shader Synthesizer

6.6.1 Indirect Shader

We use the concept of *indirect shader* to perform shader synthesis. A shader is not directly associated with the pixel. In contrast, all shaders are stored in a lookup table and the proper shader is selected at run time using a shader ID. Our shader synthesizer framework is shown in Figure 6.2. A main shader is used to resolve the shader ID and the shading process is executed on the selected shaders. The outputs from the shaders are composited in the main shader according to the shader operators discussed in section 6.3. With the support of OpenGL 2.0 [SA04], each shader is compiled into an independent shader object and linked into the main shader program.

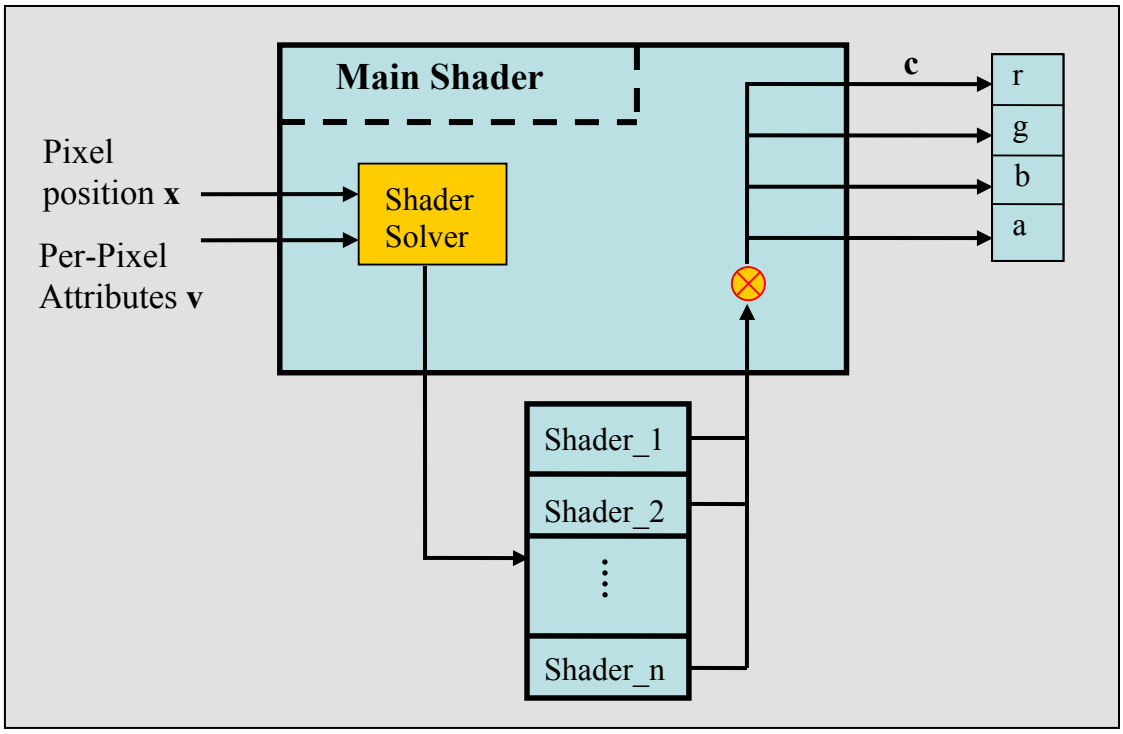


Figure 6.2: The rendering framework for indirect shader.

6.6.2 Shader Texture

A *Shader texture* stores shader IDs and other metadata information in the texture. A simple *shader texture* is a 1D lookup table storing all applicable shader IDs. A simple shader synthesizer can be implemented using a 1D *shader texture*. In the main shader, the shader solver in Figure 6.2 is simplified as a shader selector to index into the shader texture. Since the shader texture only contains the shader ID, the interpolation between the two adjacent entries in the texture is undefined and a NEAREST interpolation mode is used to locate the proper shader ID. Figure 6.3 shows a volume rendering image for a

CT head dataset. The transfer function is used as the shader selector in which the silhouette enhanced NPR shader is applied if the opacity is greater than a given threshold otherwise the DVR shader is used. This simple shader texture technique may cause a sharp transition between the visual effects from different shaders (see Figure 6.4 Top).

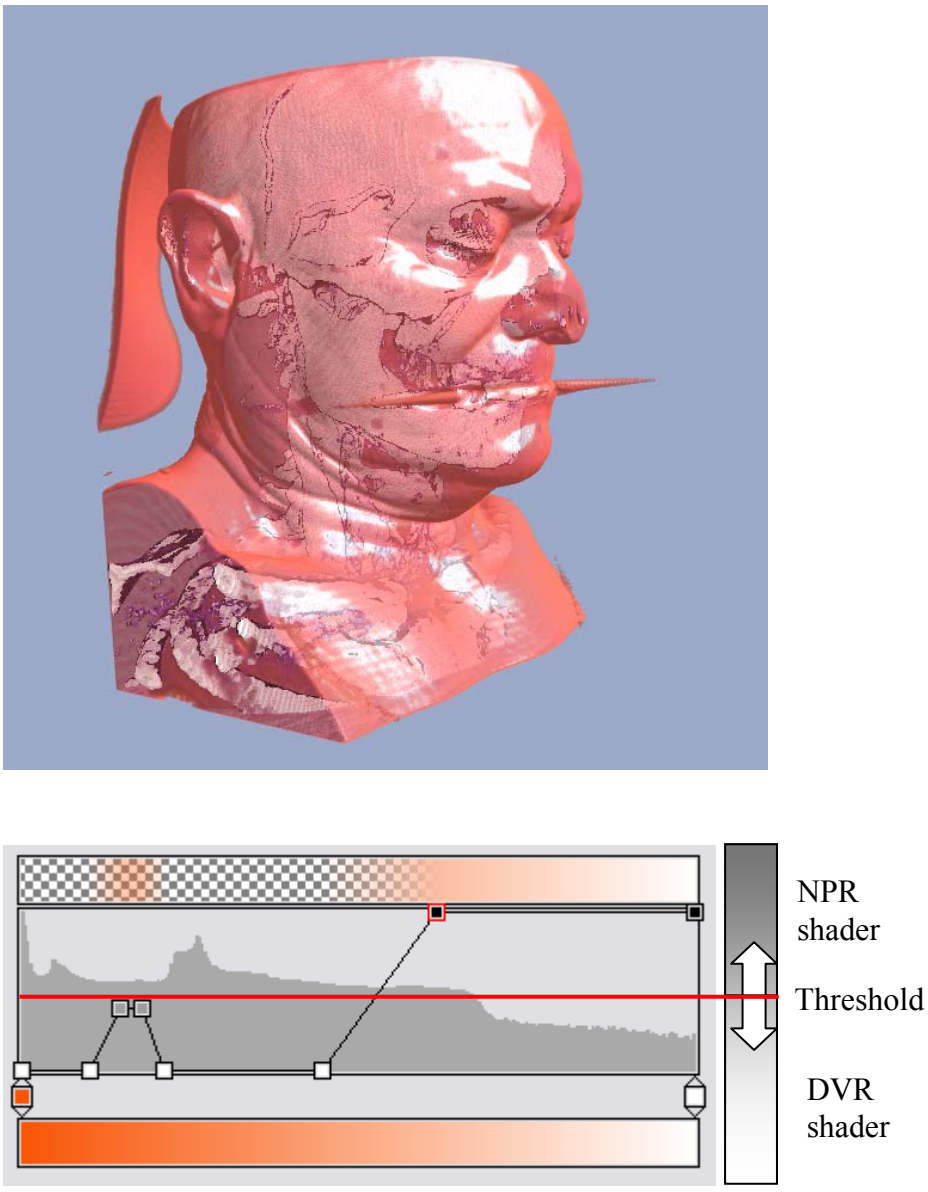


Figure 6.3: Top: Silhouette enhanced NPR shader and DVR shader for a CT Head dataset. Bottom: The transfer function is used as the shader selector.

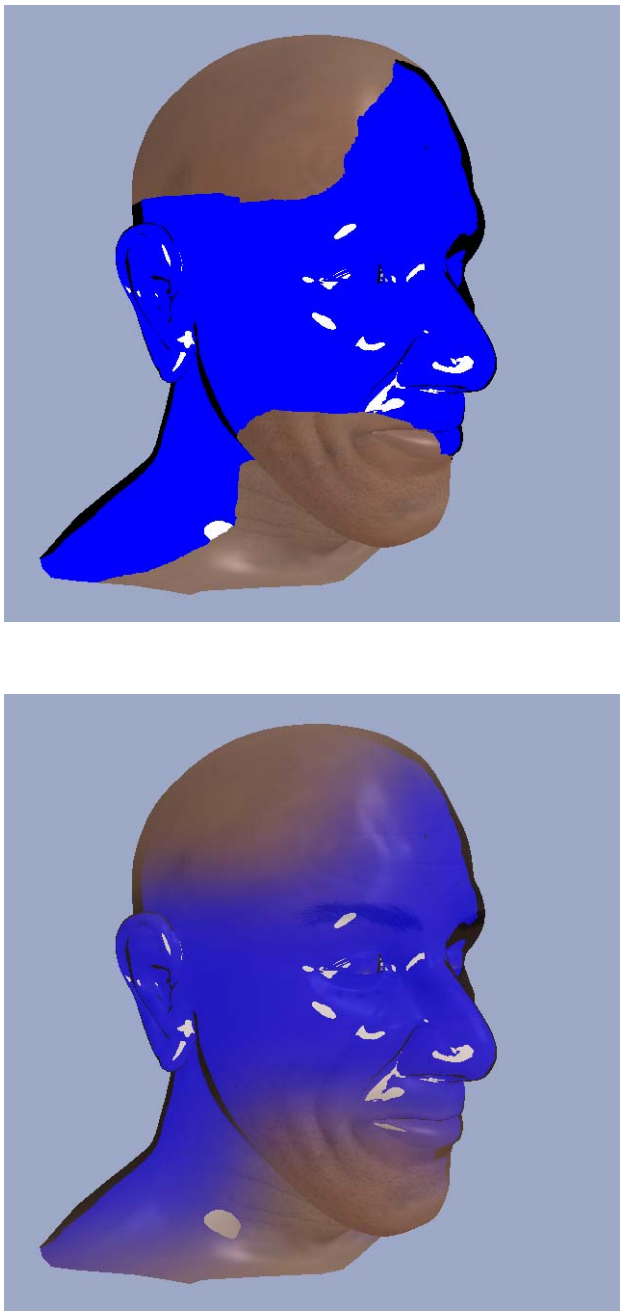


Figure 6.4: An NPR shader (toon) and a photorealistic shader (texture mapping) are rendered on the Head model. Top: A simple shader texture is used with hard edge between different shaders. Bottom: The layer-based synthesizer is used with the *over* operator to produce a smooth transition.

6.6.3 Layer-Based Shader Synthesizer

In our layer-based shader synthesizer, an image layer is generated for each shader. These image layers are then blended together. Furthermore, we extend the shader operators defined in section 6.3 to gain more visual effects.

Multi-shaders are supported in our layer-based shader synthesizer. Each texel value in the shader texture is associated with a k-tuple defining the shader attributes. Generally speaking, a higher dimension of k-tuple can provide more properties for the underlying shader; however, it is limited by the hardware texture capability. In our study, we resolve shader attributes via a triple: shader ID, shader operator ID, and an optional operator parameter. In our OpenGL implementation, this shader triple information can be encoded into a texture with the support of the integer internal format (GL_EXT_texture_integer) [OGL]. Integer texture guarantees the input integer data (texel value) will not be altered when downloaded into texture memory and fed into the shader. The integer texel value can be arbitrarily evaluated in the shader program. Figure 6.5 shows the diagram of the shader texture with four channels of a 16-bit integer. The shader ID and the shader operator ID are both encoded with 3 bits each, supporting up to 8 applicable shaders and 8 operators in the same pass rendering. The remaining 10 bits can be optionally encoded for the weight coefficients used in the *weight* operator or for other purposes. Figure 6.4 (bottom) shows a smooth transition between toon and photorealistic shaders using the **over** operator.

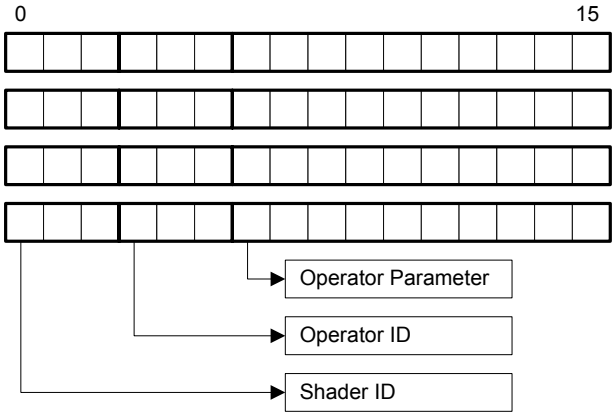


Figure 6.5: A shader texture with 4 channels. Each channel is encoded with shader ID, operator ID, and an optional parameter. This allows for the combination of 4 shaders.

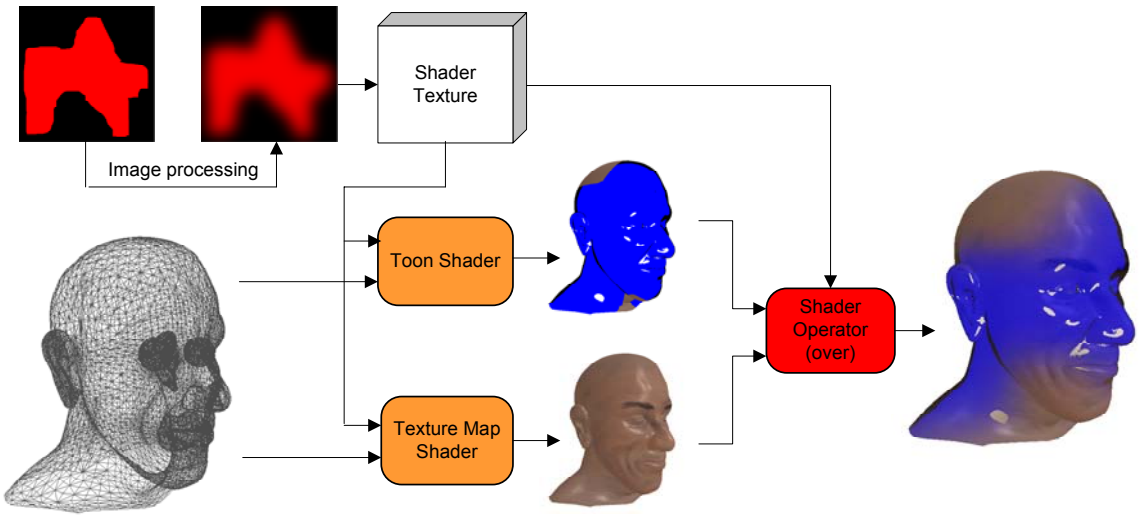


Figure 6.6: The rendering pipeline uses a rich shader texture for the Head geometry and the *over* operator is applied between two image layers.

In volume shaders, there is no real image layer since we perform the shader operations between multi-shaders for each sample along the ray cast into the volume. We can imagine that there are virtual image layers of 1x1 pixels from the different shaders for all samples along the ray and perform a shader operation between these virtual image layers for all samples. The combination of the multi-shader output is finally blended into the frame buffer according to the rendering mode.

To improve the performance, we do not run all shaders on a given input pixel. Instead, only the non-zero shader ID specified in the shader texture is executed. The shader ID of 0 is viewed as a null shader and it is simply ignored.

6.7 User-Controlled Shader Painting

Our shader synthesizer system allows the user to paint the shader onto a 2D surface or into the volume to produce an arbitrary shading effect. We discuss the techniques used in our system in this section.

6.7.1 Paint on UV-Mapping

To support user-controlled shader painting on a surface, our system draws on a shader texture with UV-mapping. The UV-mapping can be generated manually or automatically. We use the VAMP mapping [RHL] in Right Hemisphere's Deep Paint 3D to generate the UV-mapping. For simple geometry, an adaptive unwrapping technique [IC01] can be used to generate the UV-mapping directly.

Painting the shader ID onto the surface is equivalent to generate a 2D shader texture with the same as UV-Mapping, in which each texel contain the shader metadata

as shown in Figure 6.5. We use an intermediate rendering buffer to record the shader texture coordinates *(u, v)* in the UV-mapping for every pixel in the frame buffer. We call this rendering buffer an *I-buffer* only updated when painting. When drawing on the surface, the shader texture is updated from the shader painting stroke. The shader ID in the shader texture is set to the new shader ID according to the painting stroke as shown in equation 6.3.

$$ShaderTexture[I\_buffer[PixelPos]] = shaderId \tag{6.3}$$

The updated shader texture is then fed into the main shader to render the geometry with the new shaders. Figure 6.12 shows the shader ID is painted on the surface by user.

6.7.2 Volumetric Painting

6.7.2.1 Brush Stroke

Unlike surface geometry, there is no well-defined boundary inside a volumetric dataset. Instead, a transfer function is used to determine the opacity everywhere. It is difficult to determine the brush stroke depth when painting into a fuzzy volume. To solve this problem, we define the brush stroke as a ball-shaped mask with a fixed opacity value between 0 and 1 at the ball center. The opacity of the stroke ball is modulated by a three dimensional *Gaussian* function. Figure 6.7 shows the four brush strokes used in our system modulated by the different *Gaussian* functions. Note, the four strokes have the same size but with different modulated coefficients from *Gaussian* functions.

Figure 6.7: The four brush stroke balls modulated by the different *Gaussian* functions.

6.7.2.2 Brush Stroke Placement

To find out the voxels painted by the brush, the brush (ball) center is placed at the sample point in the volume matching the brush opacity. Thus all voxels covered by the brush are associated with the shader ID.

To locate the point matching the brush's center opacity, we use a ray caster to pick the matched point inside the volume. When the user paints into the volume using our painting widget, a single ray casting is sampled starting from the volume boundary. Once the first sample opacity is met with greater opacity than the stroke opacity, the ray casting is immediately stopped and the position of the sample is recorded. The voxels covered by the ball centering at the sample is associated with the shader ID of the brush stroke, as well as other information. Our method follows the instinct of "What You See Is What You Get" [HH90] in 3D painting.

For the other rendering modes, such as MIP, normally the pixel depth information is undefined. In this case, our user interface allows user to create a mask in image space. The other rendering mode can be applied inside this mask (see Figure 6.10 Bottom).

6.7.3 Brush Stroke Union

The brush stroke is generated when the surface intersection point is determined or the point inside the volume is picked by ray caster. When there are overlapped region from multiple brush strokes, care needs to be taken to compute the shader parameters for the overlapped region. To preserve the continuity, we compute the union operation (see Equation 6.4) using the algebraic sum and algebraic product to represent the shader parameters within the overlapped region.

$$A \bigcup B = a(\mathbf{x}) + b(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x}) \tag{6.4}$$

Here, $a(\mathbf{x})$ and $b(\mathbf{x})$ are the shader parameters from brush stroke A and B.

6.7.4 Painting Order

Painting order needs to be maintained for user-controlled shader painting within the overlapped region. The outputs of the shaders from shader texture will not be composited according to their order in the bit mask of the shader texture (see Figure 6.5). A painting order is specified and stored in the shader texture as shown in Figure 6.8.

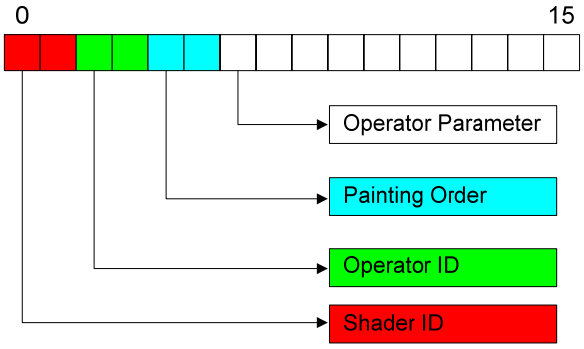


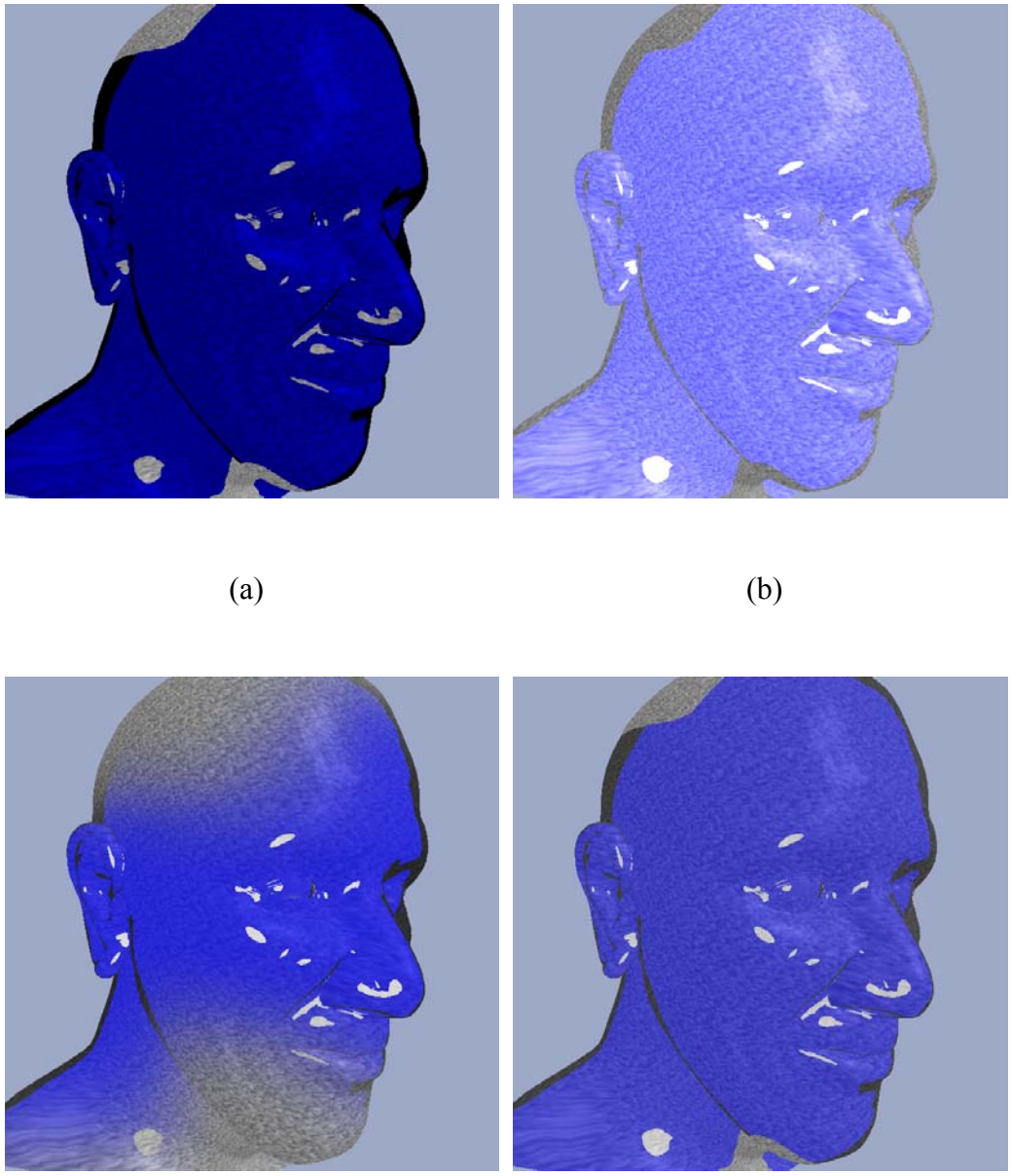
Figure 6.8: One channel of shader texture with painting order.

6.8 Experimental Results and Conclusions

All experimental images are generated on a PC with a GeForce 8800 graphics hardware and a Pentium Core 2 Q6600 2.6 GHz processor.

Figure 6.9 shows the combination between two shaders (granite and toon) using four operators: min, max, over, and weight. A smooth transition between two shaders only occurs for the *over* operator which combines the two shader results using alpha blending. Figure 6.10 top shows an image generated by a 3-layer X-Ray shader and the bottom image is generated by applying a *replace* operator between the top image and an image from a conventional X-Ray shader. Figure 6.11 shows a multi-shader rendering for volumetric datasets. A transfer function is used as a shader selector as in Figure 6.3. Figure 6.12 illustrates the user painting toon, gooch, and granite shader ID's onto the surface and they are stored in a shader texture. The surface is later rendered using the proper shaders from the shader texture. The volumetric painting with multiple shaders is shown in Figure 6.13, Figure 6.14, Figure 6.15, and Figure 6.16.

The resultant imagery indicates our indirect shader synthesizer provides a rich appearance control for the investigating data, including both surface geometries and volumetric datasets. The indirect shader synthesizer provides a novel and effective way to control the appearance of the rendering over the multi-shaders.



(a) (b)

(c) (d)

Figure 6.9: The different operators between two shaders: granite procedure shader and toon NPR shader. (a) min operator; (b) max operator; (c) over operator; (d) weight operator with both coefficients as 0.5.

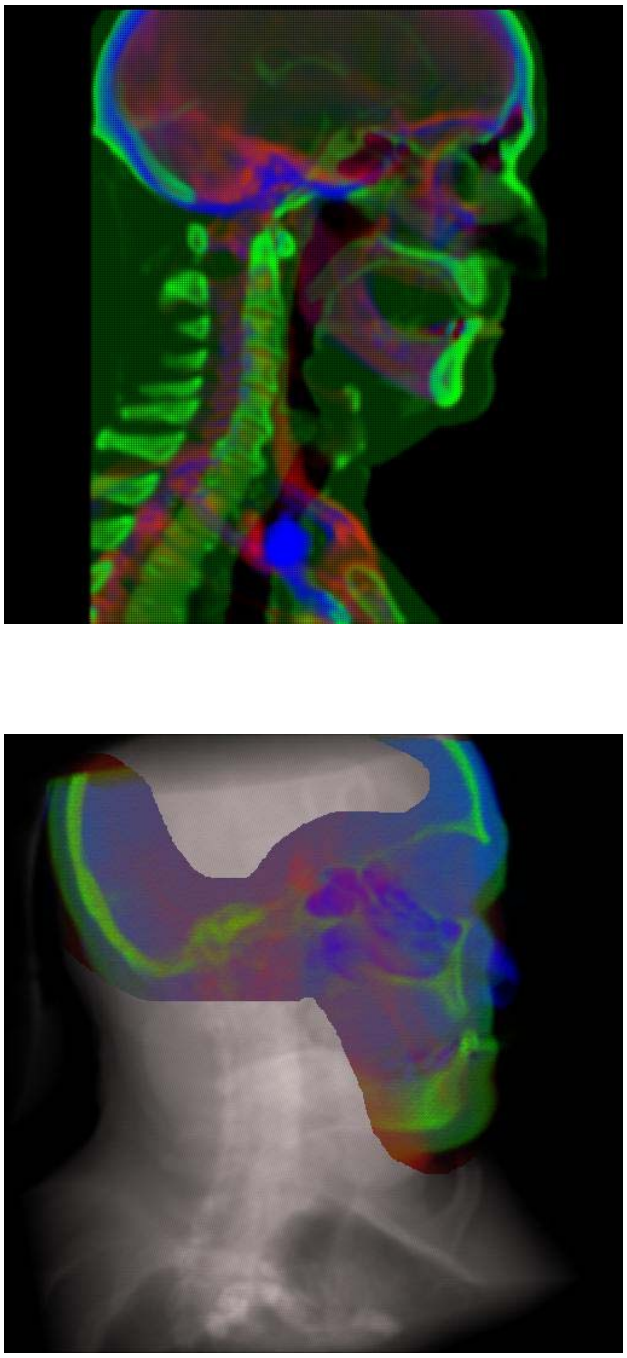
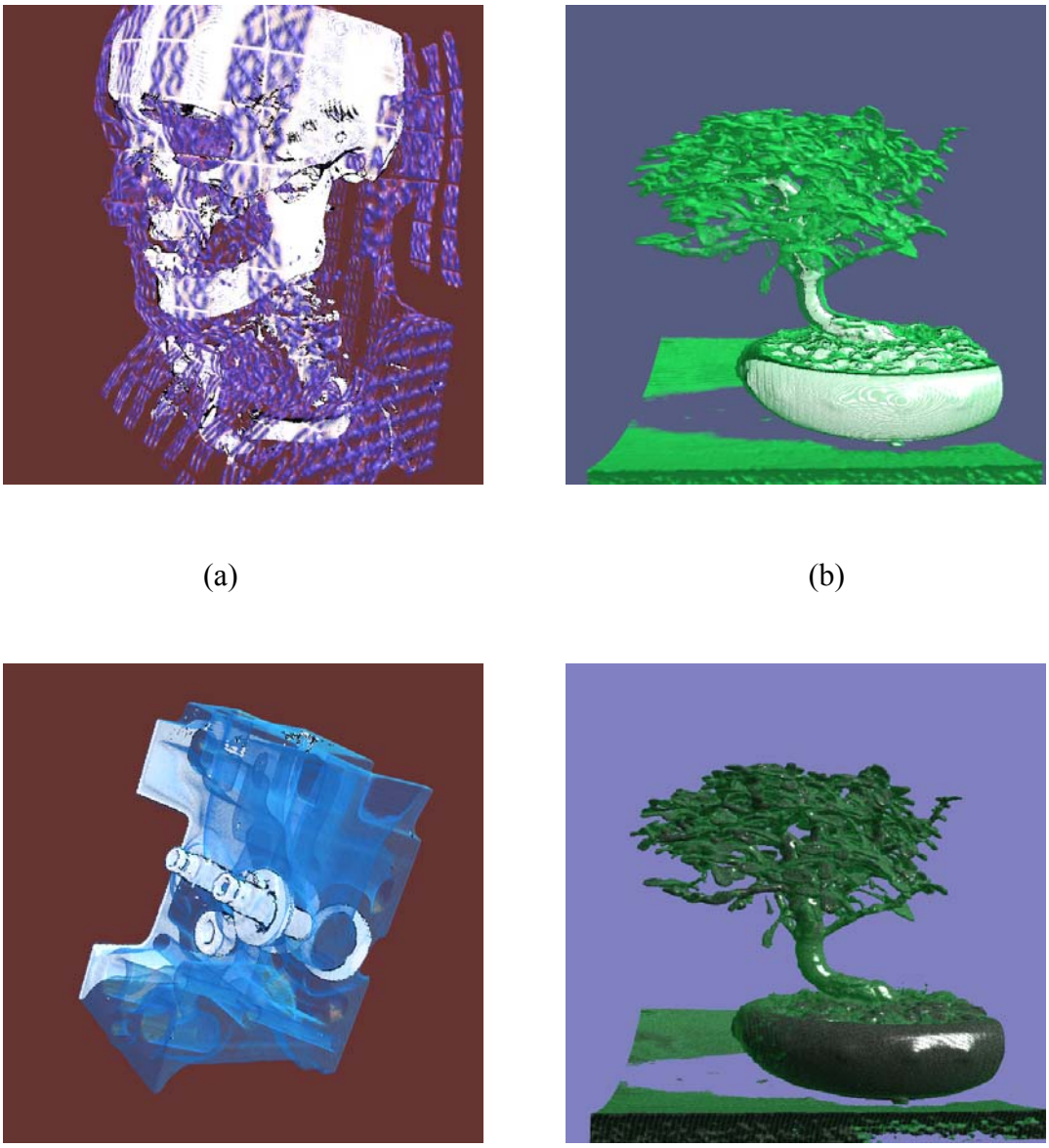


Figure 6.10: Top: The composited layer X-Ray image. Bottom: layered X-ray shader is embedded into an image space mask in the conventional X-ray image.



(a) (b)

(c) (d)

Figure 6.11: Multi-shader rendering. (a) silhouette shader + 3D brush shader. (b) silhouette shader + DVR shader for bonsai dataset. (c) silhouetter shader + DVR shader for engine dataset. (d) granite shader + DVR shader for bonsai data.

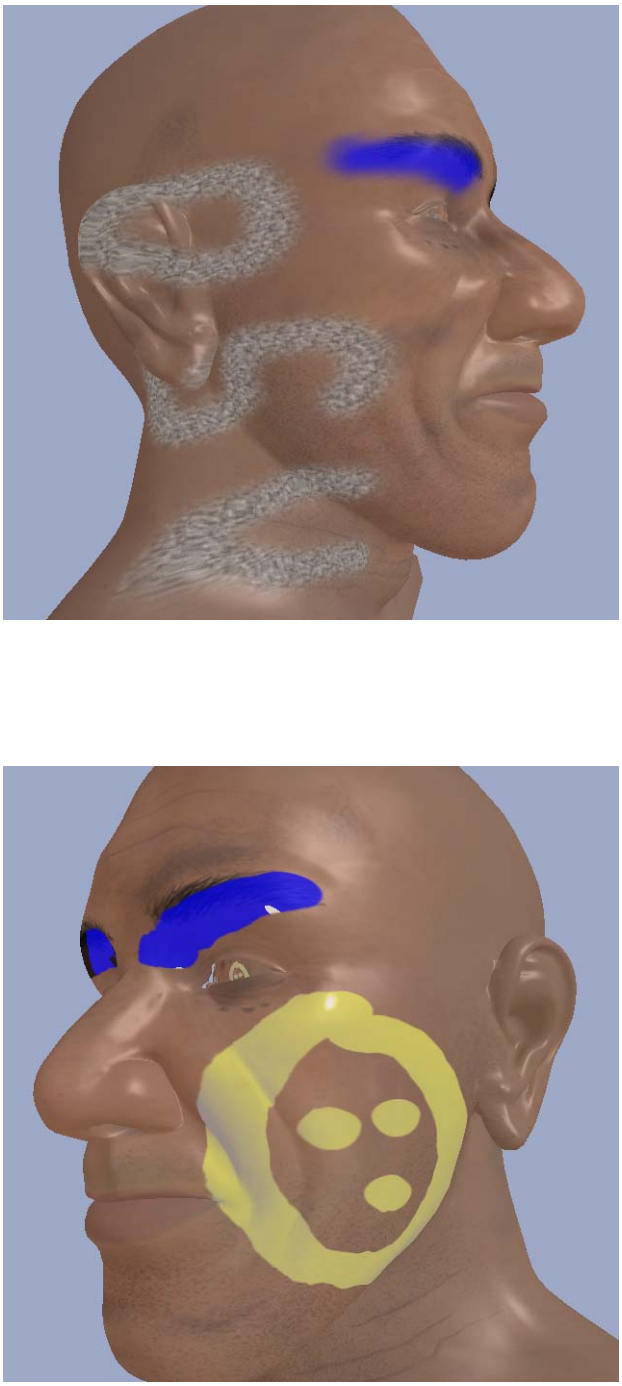


Figure 6.12: The toon, granite, and gooch shaders are drawn on the head by user.

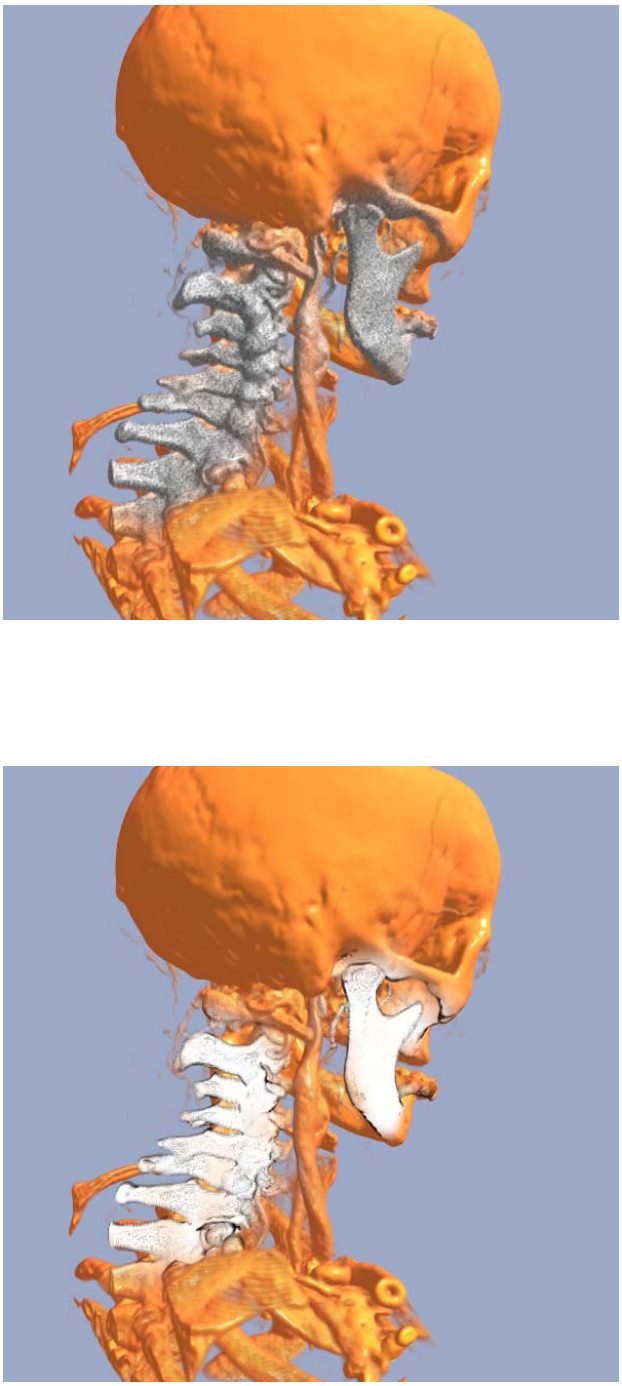


Figure 6.13: Volumetric painting with granite shader(top) and NPR shader (bottom).

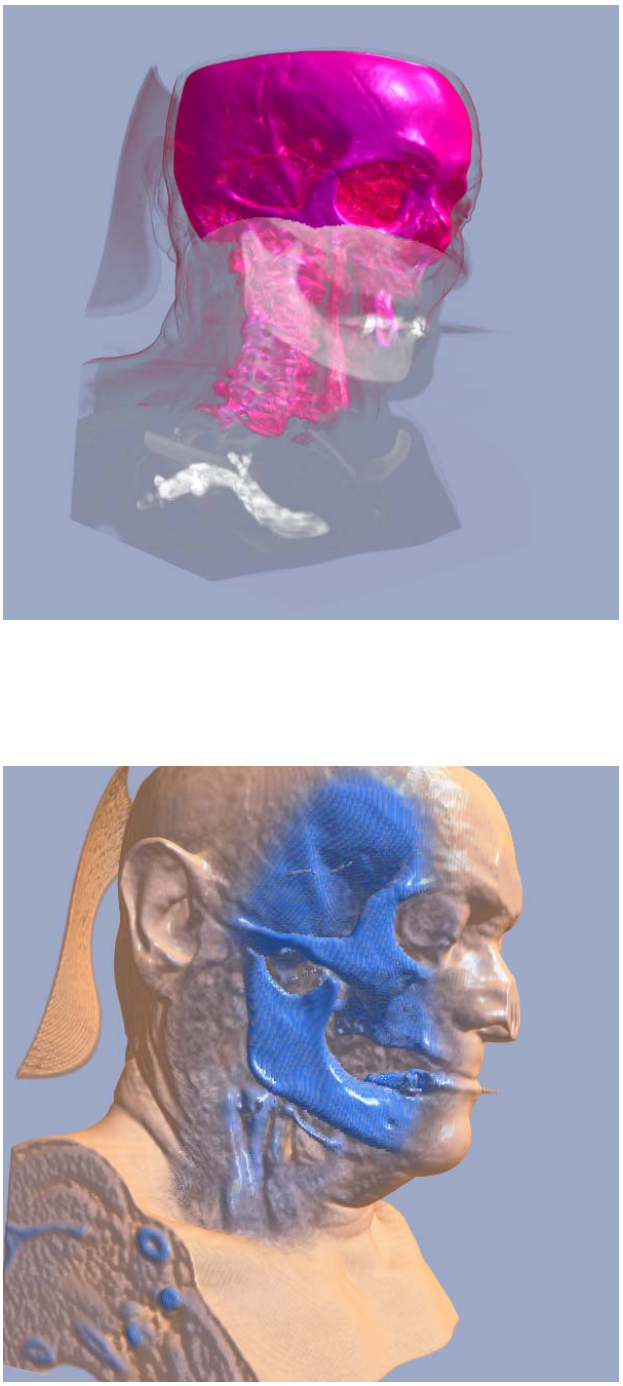


Figure 6.14: Volumetric painting with MIP shader(top) and peel shader (bottom).

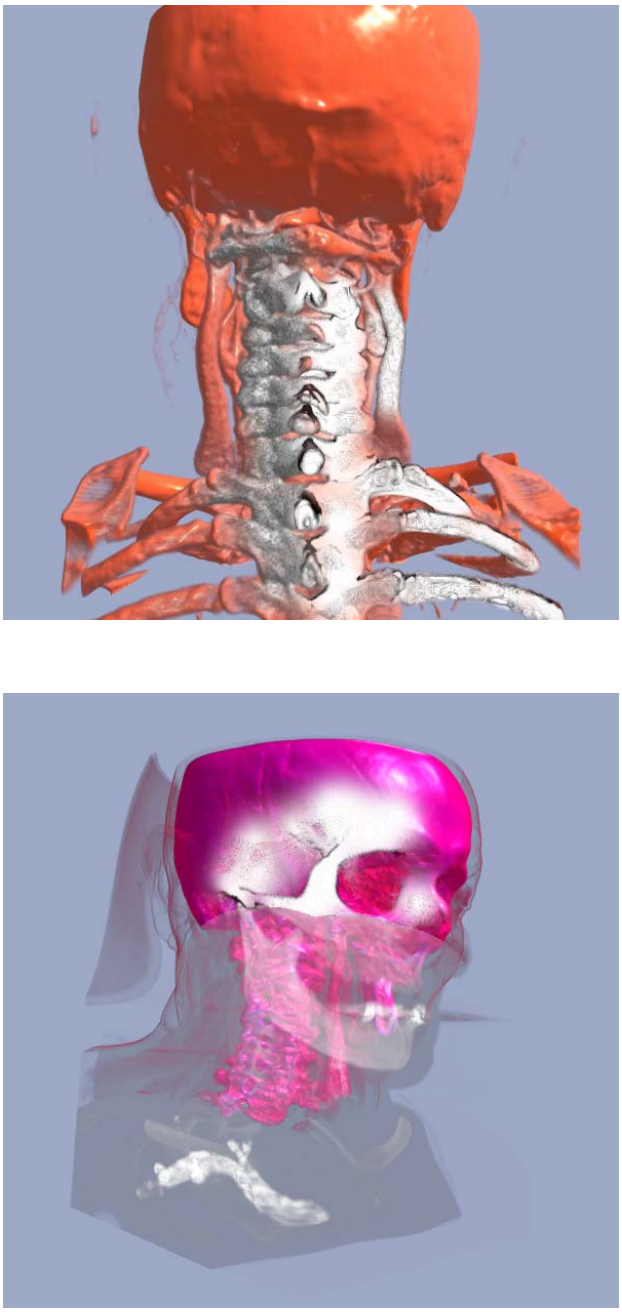


Figure 6.15: Volumetric painting with multi-shaders.

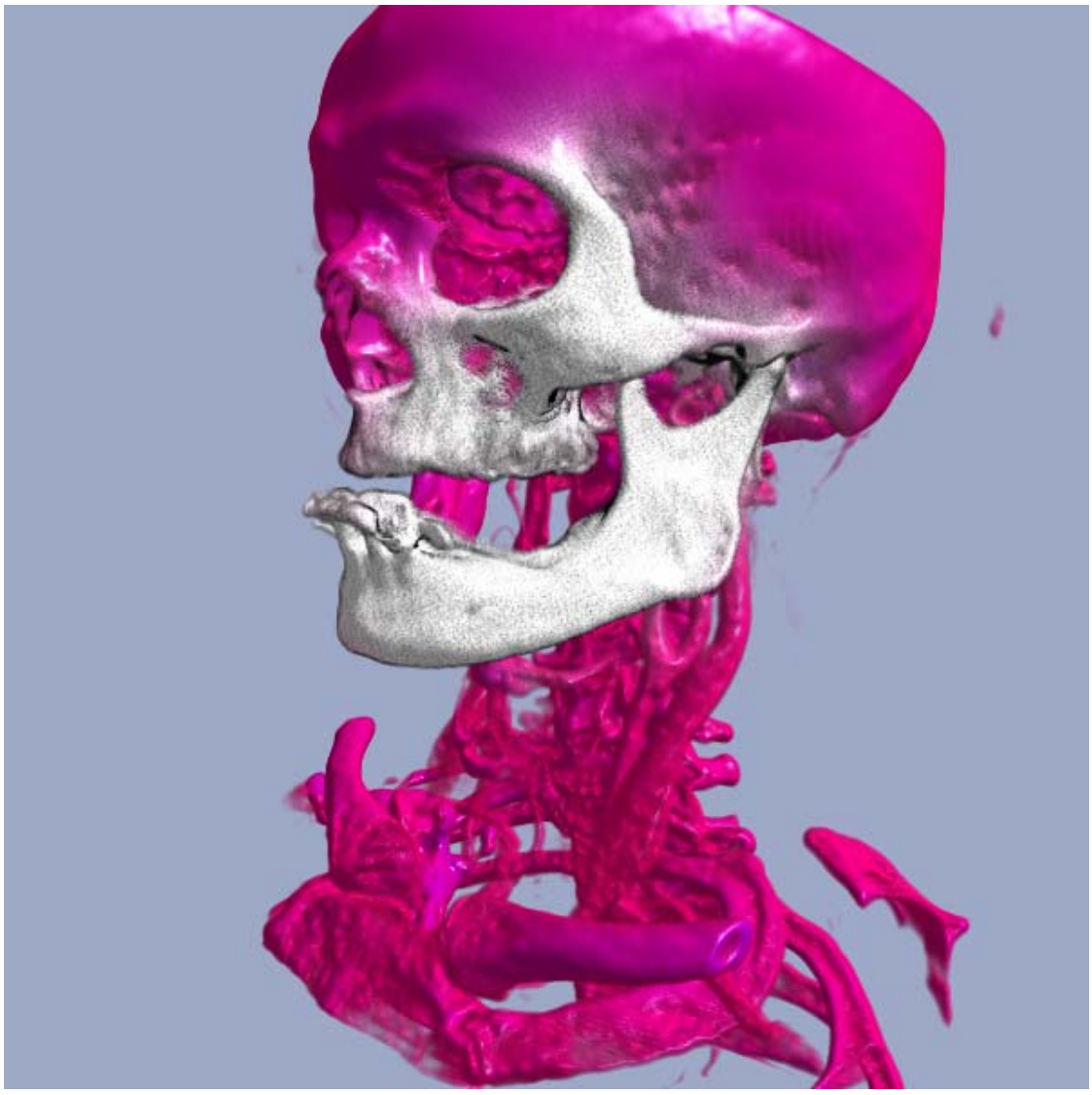


Figure 6.16: Volumetric painting with granite shader + silhouette only NPR shader.

CHAPTER 7

SUMMARY AND CONCLUSION

Graphics hardware based volume visualization techniques have been the active research topic over the last decade. The goal of this work is to investigate the graphics hardware acceleration techniques and to explore the programmable graphics hardware shaders for volume visualization. The work of this thesis has made the several contributions to the fields of computer graphics and visualization, particularly using the graphics hardware techniques.

For acceleration on volume visualization, we present a fast X-ray image generation technique using point convolution to interactively render very large volumetric datasets. Our technique can decouple the timing for X-ray image generation into the point rendering time and the point convolution time. The two stages can be further optimized and accelerated using different techniques, respectively.

The latest graphics hardware introduces the early z-culling feature. This feature allows us to setup the z-buffer of the isosurfaces segmenting the volume into empty and opaque regions. When applying slice based volume rendering (SBVR) with the pre-setup z-buffer from the isosurfaces, our technique shows the rendering performance can be

improved significantly, comparing the brute-force SBVR.

Interactive rendering and animation are the important ways to convey the dynamic information for vector field visualization. We develop several graphics hardware based volume shaders to visualize vector scalar fields. A vertex shader is presented to revisit the textured splat for fast interactive and dynamic vector field visualization. Furthermore, we generate the implicit flow fields and develop a few 3D texture based volume shaders to gain on-the-fly flow representation and appearance control.

The programmability of the graphics hardware provides many possible ways to explore the volumetric datasets and/or surface geometries. We present an indirect shader synthesis framework to composite the different shader output for the same pixel. We have shown the shader synthesizer is a well-defined tool to handle the appearance visualization for both surface geometry and volumetric data.

BIBLIOGRAPHY

[ABC*01] ALEXA, M., BEHR, J., COHEN-OR, D., FLEISHMAN, S., LEVIN, D., AND SILVA, C. T. 2001. Point Set Surfaces. In *Proc. Visualization '01*, IEEE, 21–28.

[Ake93] AKELEY, K.: Reality Engine Graphics. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27:109–116, 1993.

[ASK92] AVILA, R., SOBIERAJSKI, L. AND KAUFMAN, A.: Towards a comprehensive volume visualization system. *Proceedings of IEEE Visualization '92*, 1992.

[ATI] ATI: http://ati.amd.com/products/radeonhd4800/specs.html.

[BHR*94] BRILL, M., HAGEN, H., RODRIAN, H.-C., DJATSCHIN, W. AND KLIMENKO, S.: Streamball Techniques for Flow Visualization, In *Proc. of IEEE Visualization '94*, IEEE CS Press, 225-231, 1994.

[Bli82] BLINN, J., Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. *Proceedings of SIGGRAPH '82, Computer Graphics (16:3, 1982),* 21-29.

[BN76] BLINN, J. AND NEWELL, M.: Texture and Reflection in Computer Generated Images. *Communcations of the ACM*, 19(10):362–367, 1976.

[BWC00] BHANIRAMKA, P., WENGER, R. AND CRAWFIS, R.: Isosurfacing In Higher Dimensions, In *Proc. of IEEE Visualization 2000*, IEEE CS Press, 15-22, 2000.

[BWC04] BHANIRAMKA, P., WENGER, R. AND CRAWFIS, R.: Isosurface Construction in Any Dimension Using Convex Hulls, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 10, No. 2, pp. 130-141, 2004.

[BZX*04] BHANIRAMKA, P., ZHANG, C., XUE, D., CRAWFIS, R. AND WENGER, R.: Volume Interval Segmentation and Rendering, In *Proc. Of Volume Visualization and Graphics Symposium 2004*, pp. 55-62, Austin, TX, 2004.

[CCF94] CABRAL, B., CAM, N., AND FORAN, J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, 91–98.

[CL93] CABRAL, B., AND LEEDOM, C.: Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH '93*, ACM SIGGRAPH, 263.270, 1993.

[CM93] CRAWFIS, R. AND MAX, N.: Texture Splats for 3D Vector and Scalar Field Visualization, *Proc. Visualization'93*, IEEE CS Press, pp. 261-266, Los Alamitos, 1993.

[CN93] CULLIP, T. AND NEUMANN., U.: Accelerating volume reconstruction with 3D texture hardware. Tech. Rep. TR93-027, University of North Carolina, Chapel Hill N.C.

[CMS99] CARD, S., MACKINLAY, J., AND SHNEIDERMAN, B. 1999. Readings in Information Visualization: Using Vision to Think, Morgan Kaufmann Publishers.

[Cra96] CRAWFIS, R., Real-Time Slicing of Data Space. *Proceedings of Visualization '96 ,* pp. 271—277.

[CS03] CSEBFALVI. B., AND SZIRMAY-KALOS, L. 2003. Monte Carlo Volume Rendering. In *Proc. IEEE Visualization '03,* pp. 449-456.

[DBB06] DUTRÉ, P., BALA, K. AND BEKAERT, P.: Advanced Global Illumination, Second Edition, A K Peters Ltd., 2006.

[DCH88] DREBIN, R.A., CARPENTER, L. AND HANRAHAN, P., Volume Rendering, *Computer Graphics, SIGGRAPH'88*, 1988.

[DH92] DANSKIN, K. AND HANRAHAN, P. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, 91-98, 1992.

[DS00] Deussen, O. and Strothotte, T.: Computer-Generated Pen-and-Ink Illustration of Trees, In Proceedings of SIGGRAPH'00, July 2000.

[EKE01] ENGEL, K., KRAUS, M., AND ERTL, T.: High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics Workshop on Graphics Hardware '01*, pages 9–16. ACM SIGGRAPH, 2001.

[FMH95] FUJISHIRO, I., MAEDA, Y., AND SATO, H.: Interval volume: a solid fitting technique for volumetric data display and analysis, In *IEEE Visualization'95*, Atlanta, GA, 1995.

[EMP*03] EBERT, D., MUSGRAVE, F., PEACHEY, D., PERLIN, K., AND WORLEY, S.: *Texturing and Modeling: A Procedural Approach.* Third edition. Academic Press. 2003.

[FS97] FREUND, J. AND SLOAN, K.: Accelerated Volume Rendering Using Homogeneous Region Encoding. In *Proceedings IEEE Visualization '97*, 191-197, 1997.

[GIS03] GORLA, G., INTERRANTE, V. AND SHAPIRO, G.: Texture synthesis for 3D Shape Representation, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 9, No. 4 (Oct-Dec. 2003), pp. 217-242, 2003.

[GW92] GONZALEZ, R. AND WOODS, R. 1992. *Digital Image Processing*, Addison Wesley, pp 414 - 428.

[Har05] HARGREAVES S.: Generating shaders from hlsl fragments. In *ShaderX3: Advanced rendering with DirectX and OpenGL*, EngelW. F., (Ed.). Thomson Learning, 2005.

[HE03] HOPF, M., AND ERTL, T. 2003. Hierarchical Splatting of Scattered Data. In *Proc. IEEE Visualization '03,* pp. 433-440

[HH90] HANRAHAN, P., AND HAEBERLI, P.: Direct WYSIWYG painting and Texturing on 3D Shapes, *Computer Graphics (SIGGRAPH 90)*, Vol 24, pp. 215-223, 1990.

[Hul92] HULTQUIST, J.: Constructing stream surfaces in steady 3d vector fields. In *Proc. IEEE Visualization '92*, IEEE CS Press, 171-178, 1992.

[IC01] IGARASHI, T. AND COSGROVE, D.: Adaptive Unwrapping for Interactive Texture Painting, In *Proceedings of the 2001 symposium on Interactive 3D graphics*, 209-216, 2001.

[IK95] ITOH, T., AND KOYAMADA, K.: Automatic isosurface propagation using an extrema graph and sorted boundary cells*, IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, No. 4 (Dec. 1995), pp. 319-327, 1995.

[KCP*02] KÄHLER, R., COX, D., PATTERSON, R., LEVY, S., HEGE, H.-C., AND ABEL, T. 2002. Rendering the First Star in the Universe - A Case Study. In *Proc. Visualization' 02*, IEEE, 537–540.

[KCR99] KING, S., CRAWFIS, R. AND REID, W.: Fast Animation of Amorphous and Gaseous Volumes, *Volume Graphics '99*, Swansea, UK, pp. 336-346, 1999.

[KKH01] KNISS, J., KINDLMANN., G., AND HANSEN., C.:. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proc. IEEE Visualization '01*, IEEE CS Press, 241-248, 2001.

[KSR06] KRUGER, J., SCHNEIDER, J. AND WESTERMANN, R.: ClearView: An Interactive Conetxt Preserving Hotspot Visualization Technique. In *Proceedings of IEEE Visualization'04*, 2004.

[KvH84] KAJIYA, J. AND VON HERZEN, B.: Ray Tracing Volume Densities. In *Proc. SIGGRAPH*, 1984.

[KW03] KRÜGER, J., AND WESTERMANN, R. 2003. Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization*, Seattle, WA.

[IL95] IHM, I. AND LEE, R. K.: On Enhancing the Speed of Splatting with Indexing. *IEEE Visualization1995*, Alanta, GA, 69-76, 1995.

[LBS03] LI, G.-S., BORDOLOI, U., AND SHEN, H.-W.: Chameleon: An Interactive Texture Based Rendering Framework for Visualizing Three-Dimensional Vector Fields. In *Proc. IEEE Visualization '03*, IEEE CS Press, 241-248, 2003.

[LC87] LORENSEN, W. AND CLINE., H.: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, Vol. 21, No. 4, July, 1987.

[Lev90] LEVOY, M.: Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9,* 3(July), 245-261, 1990.

[LH91] LAUR, D. AND HANRAHAN, P.: Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, *SIGGRAPH'91*, pp. 285-288, 1991.

[Lig] http://www.lighthouse3d.com/opengl/billboarding/

[LJH03] LARAMEE, R., JOBARD, B., AND HAUSER, H.: Image Space Based Visualization of Unsteady Flow on Surfaces. In *Proc. IEEE Visualization '03*, IEEE CS Press, 131-138, 2003.

[LKM01] LINDHOLM, E., KILGARD, M. J. AND MORETON, H.: A User-Programmable Vertex Engine, *SIGGRAPH'01*, pp. 12-17, 2001.

[LL01] LEE., T.-Y. AND LIN, C.-H.: Growing-cube isosurface extraction algorithm for medical volume data. Comput Med Imaging Graph. 2001 Sep-Oct;25(5):405-15.

[LL94] LACROUTE, P. AND LEVOY, M.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform . *Comp. Graphics*, 28(4), 1994.

[LME*02] LU, A., MORRIS, C.J., EBERT, D.S., RHEINGANS, P. AND HANSEN, C.: Non-Photorealistic Volume Rendering Using Stippling Techniques. *Proceedings of IEEE Visualization'02,* 2002.

[LMK03] LI, W., MUELLER, K., AND KAUFMAN, A.: Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *IEEE Visualization*, Seattle, WA, 2003.

[LW85] LEVOY, M., AND WHITTED, T. 1985. The use of points as display primitives. *Technical report*, The University of North Carolina at Chapel Hill, Department of Computer Science.

[Mal93] MALZBENDER, T.: Fourier volume rendering. *ACM Transactions on Graphics*, Vol.12, No.3, 233–250, 1993.

[Max91] MAX, N., Sorting for Polyhedron Compositing. *Focus on Scientific*

Visualization, 1991, pp. 259-268.

[Max95] MAX, N.: Optical model for direct volume rendering, *IEEE Trans. Vis. and Comp. Graph*., vol. 1, no. 2, pp. 99-108, 1995.

[MBC93] MAX, N., BECKER, B., AND CRAWFIS, R.: Flow Volumes For Interactive Vector Field Visualization, In *Proc. of IEEE Visualization '93*, IEEE CS Press, 19-24, 1993.

[MBS*04] MAHROUS, K., BENNETT, J., SCHEUERMANN, G., HAMANN, B. AND JOY, K.: Topological Segmentation in Three-Dimensional Vector Fields, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 10, No. 2 (March 2004), pp. 198-205, 2004.

[McG05] MCGUIRE M.: The SuperShader. In *Shader X4: Advanced Rendering Techniques.* Chapter 8.1, pp. 485–498, 2005.

[MDTP*04] MCCOOL, M., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K.: Shader Algebra. In *SIGGRAPH '04*. ACM Press, pp. 787–795. 2004.

[Mic02] MICROSOFT. DirectX9 SDK. http://www.microsoft.com/DirectX. 2002

[MHB*00] MEIßNER, M., HUANG, J., BARTZ, D., MULLER, K. AND CRAWFIS, R.: A Practical Evaluation of Popular Volume Rendering Algorithms, *Proc. of Volume Vis. Sym.*, 2000.

[MKT*97] MARKOSIAN, L., KOWALSKI, M., TRYCHI, S., BOURDEV, L., GOLDSTEIN, D. AND HUGHES, J.: Real-Time Nonphotorealistic Rendering. In *Proceedings of ACM SIGGRAPH 97*, pages 113-122. 1997.

[MMC99] MUELLER, K., MOELLER, T., AND CRAWFIS, R.: Splatting without the Blur. In *Proc of IEEE Visualization '99*, IEEE CS Press, 363-370, 1999.

[MSPK06] MCGUIRE, M., STATHIS, G., PFISTER, H., AND KRISHNAMURTHI, S.: Abstract shade trees. In *Symposium on Interactive 3D Graphics and Games*, march 2006.

[NS97] NIELSON, G., AND SUNG, J.: Interval Volume Tetrahedrization, In *Proc. of IEEE Visualization '97.* IEEE CS Press, 221-228, 1997.

[NVIDIAa] NVIDIA: http://www.nvidia.com/page/geforce_6800.html.

[NVIDIAb] NVIDIA: http://www.nvidia.com/page/geforce_8800.html.

[NVIDIAc] NVIDIA: http://developer.nvidia.com/object/sdk_home.html.

[OGL] OpengGL: http://www.opengl.org/registry/

[PBS02] PARK, S., BAJAJ, C., AND SIDDAVANAHALLI, V. 2002. Case Study: Interactive Rendering of Adaptive Mesh Refinement Data. In *Proc. Visualization 02*, IEEE Computer Society Press, IEEE Computer Society, 521–524.

[PD84] PORTER, T., AND DU., T., Compositing Digital Images. *Computer Graphics (Proceedings of SIGGRAPH '84) 18:3 (1984),* 253-259.

[Per85] PERLIN, K.: An Image Synthesizer. In *Proc. SIGGRAPH*, 1985.

[PFH00] PRAUN, E., FINKELSTEIN, A. AND HOPPE, H.: Lapped textures. *Proceedings of SIGGRAPH2000*, 465-470, July 2000.

[PH04] PHARR, M. AND HUMPHREYS, G.: Physically Based Rendering: From Theory to Implementation, Morgan Kaufmann, July 2004.

[PH89] PERLIN, K. AND HOFFERT, E.: Hypertexture. In *Proc. SIGGRAPH*, 1989.

[PHF07] PLATE, J., HOLTKAEMPER, T. AND FROEHLICH, B.: A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets, In *Proceedings of IEEE Visualization'07*, 2007.

[PHK*99] PFISTER, H., HARDENBERGH, J., KNITTEL, J., LAUER, H., AND SEILER, L. 1999. The VolumePro real-time ray-casting system. *Computer Graphics (Proceedings of SIGGRAPH '99)*, 251–260.

[Pho75] PHONG, B.T.: Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[PZvBG00] PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: Surface Elements as Rendering Primitives. In *Proc. SIGGRAPH '00*, ACM, 335–342.

[REB*00] REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive Volume Rendering on Standard PC

Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *EG/SIGGRAPH Workshop on Graphics Hardware '00*, ACM, 109–118,147.

[RGW*03] ROETTGER, S., GUTHE, S., WEISKOPF, D., ERTL, T., AND STRASSER, W.: Smart Hardware-Accelerated Volume Rendering. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, 2003.

[RHL] Right Hemisphere Ltd. Deep Paint 3D (Texture Weapons), http://www.righthemisphere.com.

[RHTE99] REZK-SALAMA, C., HASTREITER, P., TEITZEL, C., AND ERTL, T.: Interactive Exploration of Volume Line Integral Convolution Based on 3D-Texture Mapping. In *Proc. of IEEE Visualization '99*. IEEE CS Press, 233-240, 1999.

[RL00] RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. SIGGRAPH '00*, ACM, 343–352.

[Ros06] ROST, R.: OpenGL(R) Shading Language, Second Edition. Addison-Wesley Professional. 2006.

[SA04] SEGAL, M. AND AKELEY, K.: The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004), online available at http://www.opengl.org/, 2004.

[SA95] SOBIERAJSKI, L. AND AVILA, R.: A Hardware Acceleration Method for Volumetric Ray Tracing. *Proceedings of IEEE Visualization '95*, 1995.

[Seb88] SEBELLA, P.: A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4): 51-58, 1988.

[SHLJ96] SHEN, H.-W., HANSEN, C., LIVNAT, Y., AND JOHNSON, C.: Isosurfacing in span space with utmost efficiency (ISSUE). *IEEE Visualization '96*, pages 287-294, 1996.

[SJM96] SHEN, H.-W., JOHNSON, C., AND MA, K.-L.: Visualizing Vector Fields Using Line Integral Convolution and Dye Advection. *Symposium on Volume Visualization'96.* IEEE Computer Society

and ACM SIGGRAPH, CA, 1996.

[SLB04] SHEN, H.-W., LI, G.-S., BORDOLOI, U.: Interactive Visualization of Three-Dimensional Vector Fields with Flexible Appearance Control, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 10, No. 4 (July 2004), pp. 434-445, 2004.

[Spi] SPITZER, J.: Texture Compositing With Register Combiners Compositing With Register Combiners, http://developer.nvidia.com/.

[SVL91] SCHROEDER, W. J., VOLPE, C. R., LORENSEN, W. E.: The Stream Polygon: A Technique for 3D Vector Field Visualization. In *Proc. IEEE Visualization '91*, IEEE CS Press, 126-132, 1991.

[TD07] TRAPP, M. AND DÖLLNER, J.: Automated Combination of Real-Time Shader Programs, In *Proceedings of Eurographics 2007*, pages 53-56, September 2007.

[TL93] TOTSUKA, T., AND LEVOY, M. 1993. Frequency Domain Volume Rendering. *Computer Graphics (Proceedings of SIGGRAPH '93*), pp 271-278.

[TNK97] TODD, J., NORMAN, F., KOENDERINK, J. AND KAPPERS, A.: Effects of Texture, Illumination, and Surface Reflectance on Stereoscopic Shape Perception*, Perception*, 26, pp. 807-822, 1997.

[TOII08] TAKAYAMA, K., OKABE, M., IJIRI, T. AND IGARASHI, T.:Lapped Solid Textures: Filling a Model with Anisotropic Textures. In *Proceedings of ACM SIGGRAPH'08*, 2008.

[Tur01] TURK, G.: Texture Synthesis on Surfaces. *Computer Graphics Proceedings (SIGGRAPH 2001)*, pp. 347-354, 2001.

[TvW03] TELEA, A., VAN WIJK., J.: 3D IBFV: Hardware-Accelerated 3D Flow Visualization. In *Proceedings of IEEE Visualization '03,* IEEE CS Press, 225-232, 2003.

[TWHS03] THEISEL, H., WEINKAUF, T., HEGE, H.-C., AND SEIDEL, H.-P. 2003. Saddle Connectors – An Approach to Visualize the Topological Skeleton of Complex 3D Vector Fields. In *Proceedings IEEE Visualization '03*, IEEE CS Press, 225-232, 2003.

[VKG04] VIOLA, I., KANITSAR, A. AND GROLLER, M: Importance-Driven Volume Rendering, In *Proceedings of IEEE Visualization'04*,

2004.

[vRHJ*04] VON RYMON-LIPINSKI, B., HANSSEN, N., JANSEN, T., RITTER, L., AND KEEVE, E. 2004. Efficient Point-Based Isosurface Exploration Using the Span-Triangle. In *Proc. IEEE Visualization '04*, pp. 441-448.

[vW93] VAN WIJK, J. J.: Implicit Stream Surfaces. In *Proc. of IEEE Visualization'93*. IEEE CS Press, 245-252, 1993.

[vW01] VAN WIJK, J. J.: Image based flow visualization. *Computer Graphics (Proc. SIGGRAPH '01),* ACM Press, 263-279, 2001.

[WE98] WESTERMANN, R., AND ERTL, T.: Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. of SIGGRAPH ' 98*, ACM Press, 169-177, 1998.

[Wes89] WESTOVER, L. 1989. Interactive Volume Rendering, *Proc. of the Chapel Hill Workshop on Volume Visualization,* C. Upson, ed., pp. 9-16, Chapel Hill, NC, May, 1989.

[Wes90] WESTOVER, L. A.: Footprint Evaluation for Volume Rendering, *Computer Graphics (Proceedings of SIGGRAPH)*, 24(4), pp. 367–376, August 1990.

[WFP*01] WAND, M., FISCHER, M., PETER, I., MEYER AUF DER HEIDE, F., and STRASSER, W.: The Randomized z-Buffer Algorithm. In *Proc. SIGGRAPH '01*, ACM, 361–370, 2001.

[Wil92] WILLIAMS, P. Visibility Ordering of Meshed Polyhedra. In *ACM Transactions on Graphics*, 11 (4), 103-126, April 1992.

[WJE00] WESTERMANN, R., JOHNSON, C., AND ERTL, T.: A Level-Set Method for Flow Visualization. In *Proc. of IEEE Visualization 2000*, IEEE CS Press, 147-154, 2000.

[WKE02] WEILER, M., KRAUS, M. AND ERTL, T.: Hardware-Based View-Independent Cell Projection. In *Symposium on Volume Visualization'02*, pp. 13-22, Boston, MA, 2002.

[WLMK02] WEI, X., LI, W., MUELLER, K. AND KAUFMAN, A.: Simulating Fire with Texture Splats, *IEEE Visualization 2002*, pp. 227-237, Boston, MA, 2002.

[WMFC02] WYLIE, B., MORELAND, K., FISK, L. A. AND CROSSNO, P.:

Tetrahedral projection using Vertex Shaders. In *Symposium on Volume Visualization'02*, Boston, MA., pp. 7-12, 2002.

[WvG91] WILHELMS, J. AND VAN GELDER, A.: A coherent projection approach for direct volume rendering, *Cmputer Graphics*, vol. 25, no. 4, pp. 275-284, 1991.

[Wyn] WYNN, C.: OpenGL Vertex Programming on Future-Generation GPUs, http://developer.nvidia.com/.

[XC03] XUE, D., AND CRAWFIS, R. 2003. Efficient Splatting Using Modern Graphics Hardware, *Journal of Graphics Tools,* Vol 8. No. 3, pp. 1-21.

[XC04] XUE, D. AND CRAWFIS, R.: Fast Dynamic Flow Volume Rendering Using Textured Splats on Modern Graphics Hardware, *Proceedings of SPIE EI 2004*, pages 133-140, San Jose, CA, 2004.

[XZC04] XUE, D., ZHANG, C. AND CRAWFIS, R.: Rendering Implicit Flow Volumes, *Proceedings IEEE Visualization'04*, Austin, TX, 2004.

[XZC05] XUE, D., ZHANG, C., AND CRAWFIS, R. 2005. iSBVR: Isosurface-aided Hardware Acceleration Techniques for Slice-Based Volume Rendering. In *Proc. Of International Workshop on Volume Graphics*.

[YS93] YAGEL, R. AND SHI, Z.: Accelearted Volume Animation by Space-Leaping. In P*roceedings IEEE Visualization '93*, 62-69.

[ZPvBG01] ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface Splatting. In *Proc. SIGGRAPH '01*, ACM, 371–378.

[ZSH96] ZÖCKLER, M., STALLING, D., AND HEGE, H.-C.: Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proc. of Visualization '96*, IEEE CS Press, 107-114, 1996.